PROGRAMACION DEL 6502

RODNAY ZAKS



PROGRAMACIÓN DEL 6502

RODNAY ZAKS

PROGRAMACION DEL 6502



marcombo
BOIXAREU EDITORES

BARCELONA-MEXICO

Título de la edición original

PROGRAMMING THE 6502
por Rodnay Zaks



Ilustración de cubierta por Daniel Le Noury
Traducción al castellano por Luis Joyanes Aguilar

ISBN: 84-267-0616-9
ISBN: 0-89588-135-7, SYBEX Inc., edición original
Depósito legal: B. 9335 - 1986
Impreso en España
Printed in Spain
Imprenta Juvenil, S. A. - Maracaibo, 11 - 08030 Barcelona

Índice general

Prólogo

Este libro ha sido concebido como un texto completo y autodidáctico para aprender a programar, utilizando el 6502. Puede ser leído por una persona que nunca haya programado antes de ahora y será de interés para toda persona que utilice el 6502.

Para la persona que ya tiene experiencia en programación, este libro le enseñará las técnicas de programación utilizando (o basándose en) las características específicas del 6502. Este libro abarca las técnicas elementales, e intermedias, necesarias para comenzar a programar eficazmente.

Este texto tiene por finalidad proporcionar un verdadero nivel de competencia al lector que desee programar empleando este microprocesador. Naturalmente, ningún libro puede enseñar la programación eficazmente, a no ser que se practique realmente. No obstante, se confía en que este libro lleve al lector al punto en que pueda comenzar a programar por sí mismo y resolver problemas sencillos, o moderadamente complejos, empleando un microordenador.

Este libro se basa en la experiencia del autor, quien ha enseñado la programación de microordenadores a más de 1000 personas, en consecuencia, el libro está sólidamente estructurado. Los capítulos van, normalmente, desde lo sencillo a lo complejo. Los lectores que hayan aprendido ya programación elemental se pueden saltar el capítulo de introducción. Para los que nunca hayan programado, las últimas secciones de algunos capítulos pueden exigir una segunda lectura. El libro ha sido concebido para conducir al lector sistemáticamente, a través de todos los conceptos y técnicas requeridos, hasta la elaboración de programas de complejidad creciente. Se recomienda, pues, encarecidamente, que se siga el orden de los capítulos. Además, para obtener resultados efectivos, es importante que el lector trate de

resolver tantos ejercicios como sea posible. La dificultad en los ejercicios ha sido graduada cuidadosamente. Han sido concebidos para verificar que las nociones que han sido presentadas se han comprendido realmente. Si no se hacen los ejercicios de programación, no será posible aprovechar completamente el valor didáctico de este libro. Ciertos ejercicios pueden requerir tiempo, como, por ejemplo, el ejercicio de la multiplicación. Sin embargo, haciendo los ejercicios, se programará realmente y *se aprenderá por la práctica,* lo cual resulta indispensable.

El contenido de este libro ha sido comprobado cuidadosamente y se cree es fiable. No obstante, inevitablemente, algunos errores tipográficos o de otra clase se podrán encontrar. El autor agradecerá cualquier comentario de los lectores de modo que las futuras ediciones puedan beneficiarse de su experiencia. Será bien acogida cualquier otra sugerencia de mejora, tal como otros programas deseados, desarrollados o encontrados de valor por los lectores.

PRÓLOGO A LA CUARTA EDICIÓN EN INGLÉS

En los cinco años transcurridos desde que fue publicado por primera vez este libro, ha aumentado exponencialmente el número de usuarios del microprocesador 6502 y continúa creciendo. Este libro se ha expandido con ellos.

El tamaño de la segunda edición ha aumentado casi 100 páginas, habiendo sido añadido la mayoría de material nuevo en los capítulos 1 y 9. A lo largo del libro se han introducido mejoras constantemente. En esta cuarta edición se han incluido, en el apéndice I, las soluciones de los ejercicios, a petición de muchos lectores que deseaban cerciorarse de que su conocimiento de la programación del 6502 era completamente satisfactorio.

Deseo expresar mi agradecimiento a los muchos lectores de las ediciones anteriores que han contribuido a mejorarlas con valiosas sugerencias. Debo un especial reconocimiento a Eric Novikoff y Chris Williams por su colaboración en las soluciones de los ejercicios, así como por los complicados ejemplos de programación del capítulo 9. También expreso mi agradecimiento especialmente a Daniel J. David por las mejoras que ha sugerido. También se deben varias modificaciones y enriquecimientos al análisis y los valiosos comentarios de Philip K. Hooper, John Smith, Ronald Long, Charles Curlay, N. Harris, John McClenon, Douglas Trusty, Fletcher Carson y el Profesor Myron Calhoun.

RECONOCIMIENTOS

El autor desea expresar su estimación y reconocimiento a Rockwell International y, en particular, a Scotty Maxwell, quien le facilitó uno de los primeros sistemas de desarrollo de la serie 65. La disponibilidad de esta potente herramienta cuando escribió la primera versión de este libro fue la principal ayuda para la exacta y eficiente verificación de todos los programas. Asimismo mi gratitud al Profesor Myron Calhoun por su colaboración.

1 Conceptos básicos

INTRODUCCIÓN

En este capítulo se introducirán los conceptos básicos relativos a programación de ordenadores. El lector ya familiarizado con estos conceptos puede sentir la tentación de ojear rápidamente el contenido de este capítulo y pasar al capítulo 2. Sin embargo, incluso al lector experimentado se le recomienda que lea este capítulo de introducción. Se presentan en el mismo muchos conceptos importantes, incluyendo, por ejemplo, el complemento a dos, BCD y otras representaciones. Algunos de estos conceptos podrán ser nuevos para el lector; otros pueden mejorar el conocimiento y las aptitudes de programadores experimentados.

¿QUÉ ES LA PROGRAMACIÓN?

Planteado un problema, se debe idear, en primer lugar, una manera de resolverlo que, expresada como procedimiento paso a paso, se denomina *algoritmo.* Un algoritmo es una especificación paso a paso de la resolución de un problema dado. Debe terminar en un número finito de pasos. Este algoritmo se puede expresar en cualquier lenguaje o simbolismo. Un ejemplo sencillo de un algoritmo es:

1 — meter la llave en el agujero de la cerradura
2 — girar la llave una vuelta completa a la izquierda
3 — agarrar el picaporte
4 — girar el picaporte a izquierda y empujar la puerta

En este momento, si el algoritmo es correcto para el tipo de cerradura considerada, se abrirá la puerta. Este procedimiento de cuatro pasos se considera un algoritmo de apertura de puerta.

Una vez que la resolución de un problema se ha expresado en la forma de un algoritmo, éste se debe ejecutar por el ordenador. Lamentablemente, es un hecho bien conocido, actualmente, que los ordenadores no pueden comprender ni ejecutar un programa expresado en castellano (o cualquier otro lenguaje humano). La razón reside en la *ambigüedad de la sintaxis* de todos los lenguajes humanos comunes. Únicamente un subconjunto bien definido del lenguaje natural puede ser "comprendido" por el ordenador. Esto se denomina *lenguaje de programación.*

Se denomina *programación* a la conversión de un algoritmo en una secuencia de instrucciones de un lenguaje de programación. Para ser más preciso, la fase de traducción propiamente dicha del algoritmo a un lenguaje de programación se llama *codificación.* La programación designa realmente no sólo la codificación, sino también la concepción de los programas y "estructuras de datos" que realizarán el algoritmo.

La programación efectiva requiere no solamente comprender las técnicas posibles de puesta en práctica ("implementación") de algoritmos estándar, sino también la explotación inteligente de todos los recursos de hardware del ordenador, tales como los registros internos, memoria y dispositivos periféricos, además de una creatividad en la utilización de las estructuras de datos. Estas técnicas se tratarán en los capítulos siguientes.

La programación exige también una disciplina estricta de la documentación, de modo que los programas sean comprensibles por otras personas, de igual modo que por el autor. La documentación debe ser a la vez interior y exterior al programa.

La documentación interna al programa consiste en comentarios incorporados en el cuerpo de un programa, que explican su funcionamiento.

La documentación externa se refiere a los documentos de concepción que están separados del programa: explicaciones escritas, manuales y diagramas de flujo.

DIAGRAMAS DE FLUJO

Casi siempre se utiliza un paso intermedio entre el *algoritmo* y el *programa.* Se denomina *diagrama de flujo.* Un diagrama de flujo es simplemente una representación del algoritmo expresada como una secuencia de rectángulos y de rombos que contienen las etapas del algoritmo. Los rectángulos se utilizan para *órdenes* (mandatos) o "instrucciones ejecutables". Los rombos se utilizan para *pruebas* tales como: Si la información X es cierta, enton-

ces realice la acción A, y si no lo es, realizar B. En lugar de explicar, en este momento, una definición formal de los diagramas de flujo, se introducirán y comentarán los mismos más adelante cuando se expliquen los programas en el libro.

El diagrama de flujo es una etapa intermedia muy aconsejable entre la especificación del algoritmo y la codificación propiamente dicha de la resolución. Es significativo que se haya constatado que quizás solamente el 10 % de los programadores son capaces de escribir un programa correcto sin tener que realizar el diagrama de flujo. Lamentablemente, se ha observado también que el 90 % de los programadores piensan que pertenecen a este 10 %. Como resultado de ello, el 80 % de estos programas, por término medio, no funcionarán la primera vez que se ejecuten en el ordenador. (Estos porcentajes, naturalmente, no pretenden ser precisos). En resumen, la mayoría de los programadores principiantes ven raramente la necesidad de trazar un diagrama de flujo. Esto suele conducir a programas "no limpios" o erróneos. Deben entonces emplear mucho tiempo en comprobar y corregir su programa (esto se llama fase de depuración o "*debugging*"). En consecuencia, es muy aconsejable, en todos los casos, tener la disciplina de trazar el diagrama de flujo. Ello requiere un pequeño tiempo adicional antes de la codificación, pero resultará habitualmente un programa limpio que se ejecutará

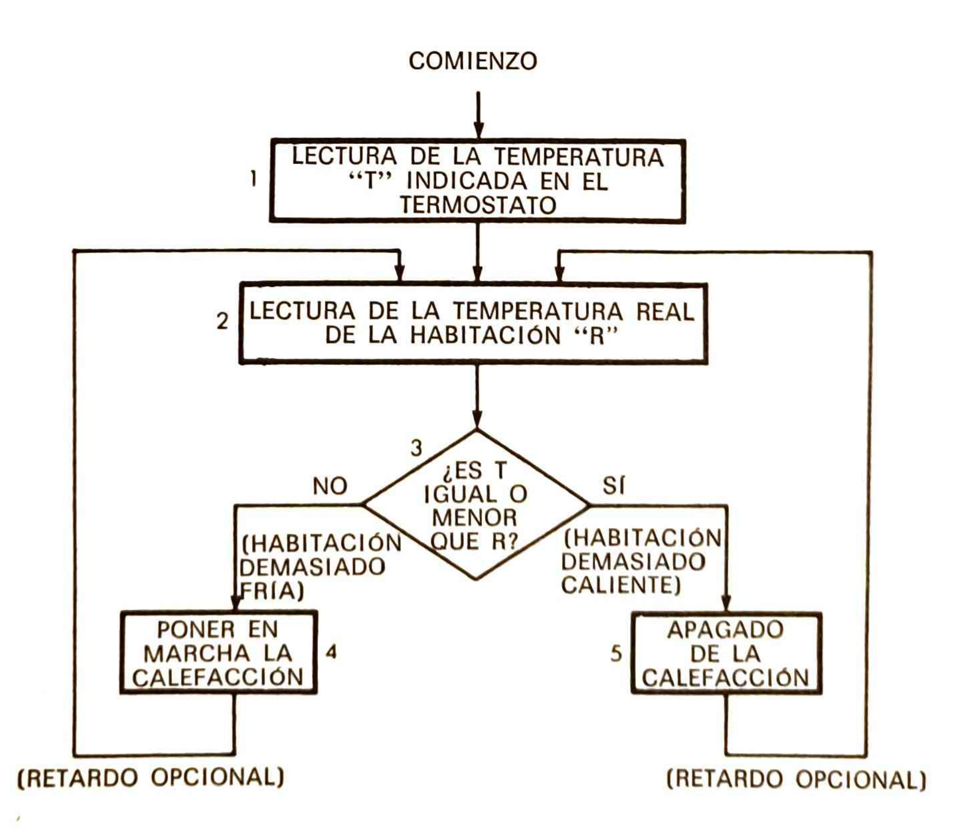


Figura 1-1 Diagrama de flujo para mantener constante la temperatura de una habitación.

correcta y rápidamente. Cuando los diagramas de flujo están bien comprendidos, un pequeño porcentaje de programadores podrán realizar esta etapa mentalmente sin tener que hacerlo en el papel. Desgraciadamente, en tales casos, los programas que escriben suelen ser difíciles de comprender por cualquier otro programador sin la documentación proporcionada por el diagrama de flujo. Por consiguiente, se recomienda universalmente atenerse estrictamente a la disciplina de trazar el diagrama de flujo para todo programa importante. Se proporcionarán numerosos ejemplos a lo largo del libro.

REPRESENTACIÓN DE LA INFORMACIÓN

Todos los ordenadores manejan la información en forma de números o en forma de caracteres. Examinemos en este apartado las representaciones interna y externa de la información en un ordenador.

REPRESENTACIÓN INTERNA DE LA INFORMACIÓN

En un ordenador toda la información se almacena como grupos de bits. Un *bit* es la abreviatura de un *dígito binario* ("0" o "1"). A causa de las limitaciones de la electrónica tradicional, la única manera práctica de representar informaciones es utilizar la lógica de dos estados (la representación de los estados "0" y "1"). Los dos estados de los circuitos utilizados en electrónica digital suelen ser "conexión" (on) o "desconexión" (off) y se representan lógicamente por los símbolos "0" o "1". Ya que estos circuitos se utilizan para realizar funciones "lógicas", se denominan "lógicos binarios". Resulta de ello que, prácticamente, todo el proceso de la información se realiza actualmente en formato binario. En el caso de los microprocesadores en general, y en el 6502 en particular, estos bits se estructuran en grupos de ocho. Un grupo de ocho bits se denomina *byte* (octeto). Un grupo de cuatro bits se denomina *nibble* (cuaterna).

Examinemos ahora cómo se representa internamente la información en el formato binario. Dos entidades se deben representar dentro del ordenador. La primera es el programa, que es una secuencia de instrucciones. La segunda son los datos con los que funcionará el programa, que pueden incluir números o texto alfanumérico. Comentaremos a continuación tres representaciones: programa, números y caracteres alfanuméricos.

Representación del programa

Todas las instrucciones se representan internamente por uno o varios bytes. Las instrucciones denominadas "cortas" se representan por un solo

byte. Una instrucción larga se representará por dos o más bytes. Ya que el 6502 es un microprocesador de ocho bits, busca y carga bytes sucesivamente de su memoria. Por tanto, una instrucción de un solo byte tiene siempre un modo potencial para ejecución más rápida que una instrucción de dos o tres bytes. Se verá más adelante que ello constituye una característica importante del juego de instrucciones de cualquier microprocesador y en particular del 6502, en donde se ha realizado un esfuerzo especial para proporcionar tantas instrucciones de un solo byte como sea posible, con el fin de mejorar la eficacia de la ejecución del programa. Sin embargo, la limitación de la longitud a 8 bits ha traído consigo importantes restricciones, las cuales serán bosquejadas más adelante. Este es un ejemplo clásico de solución de compromiso entre la velocidad y la flexibilidad en la programación. El código binario utilizado para representar instrucciones es impuesto por el fabricante. El 6502, como cualquier otro microprocesador, se suministra con un juego fijo de instrucciones. Estas instrucciones se definen por el fabricante y se indican al final de este libro, con su código. Cualquier programa se expresará como una secuencia de estas instrucciones binarias. Las instrucciones del 6502 se presentan en el capítulo 4.

Representación de datos numéricos

La representación de números no es tan evidente y se pueden distinguir varios casos. Se deben representar primeramente los enteros y después los números con signos (esto es, números positivos y negativos) y, finalmente, debemos poder representar números decimales. Examinemos ahora estos requisitos y las soluciones posibles.

La representación de enteros se puede realizar utilizando una forma *binaria directa.* La representación binaria directa no es más que la representación del valor decimal de un número en el sistema binario. En el sistema binario, el bit más a la derecha (menos significativo) representa 2 elevado a 0. El siguiente, inmediatamente a la izquierda, representa 2 a la potencia 1, el siguiente representa 2 a la potencia 2 y el bit más significativo situado más a la izquierda, representa 2 elevado a 7 = 128.

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

representa

$$b_72^7 + b_62^6 + b_52^5 + b_42^4 + b_32^3 + b_22^2 + b_12^1 + b_02^0$$

Las potencias de 2 son:

$2^7 = 128,\ 2^6 = 64,\ 2^5 = 32,\ 2^4 = 16,\ 2^3 = 8,\ 2^2 = 4,\ 2^1 = 2,\ 2^0 = 1$

La representación binaria es análoga a la representación decimal de números, en donde "123" representa:

$$
\begin{array}{rrcr}
 & 1 \times 100 & = & 100 \\
+ & 2 \times 10 & = & 20 \\
+ & 3 \times 1 & = & 3 \\
\hline
 & & = & 123
\end{array}
$$

Obsérvese que $100 = 10^2$, $10 = 10^1$, $1 = 10^0$.
En esta "notación posicional", cada dígito representa una potencia de 10. En el sistema binario, cada dígito binario, o "bit", representa una potencia de 2, en vez de una potencia de 10 en el sistema decimal.

Ejemplo: "00001001" en binario representa:

$$
\begin{array}{lrcll}
 & 1 \times 1 & = & 1 & (2^0) \\
 & 0 \times 2 & = & 0 & (2^1) \\
 & 0 \times 4 & = & 0 & (2^2) \\
 & 1 \times 8 & = & 8 & (2^3) \\
 & 0 \times 16 & = & 0 & (2^4) \\
 & 0 \times 32 & = & 0 & (2^5) \\
 & 0 \times 64 & = & 0 & (2_6) \\
 & 0 \times 128 & = & 0 & (2_7) \\
\cline{2-4}
\text{en decimal:} & & = & 9 &
\end{array}
$$

Examinemos algunos ejemplos más:

"10000001" representa:

$$
\begin{array}{lrcr}
 & 1 \times 1 & = & 1 \\
 & 0 \times 2 & = & 0 \\
 & 0 \times 4 & = & 0 \\
 & 0 \times 8 & = & 0 \\
 & 0 \times 16 & = & 0 \\
 & 0 \times 32 & = & 0 \\
 & 0 \times 64 & = & 0 \\
 & 1 \times 128 & = & 128 \\
\cline{2-4}
\text{en decimal:} & & = & 129
\end{array}
$$

"10000001" representa, por consiguiente, el número decimal 129.

Examinando la representación binaria de números, se comprenderá por qué los bits están numerados del 0 al 7, de derecha a izquierda. El bit 0 es "b_0" y corresponde a 2^0. El bit 1 es "b_1" y corresponde a 2^1, y así sucesivamente.

Los equivalentes binarios de los números de 0 a 255 se muestran en la figura 1-2.

Decimal	Binario	Decimal	Binario
0	00000000	32	00100000
1	00000001	33	00100001
2	00000010	•	
3	00000011	•	
4	00000100	•	
5	00000101	63	00111111
6	00000110	64	01000000
7	00000111	65	01000001
8	00001000	•	
9	00001001	•	
10	00001010	127	01111111
11	00001011	128	10000000
12	00001100	129	10000001
13	00001101		
14	00001110	•	
15	00001111	•	
16	00010000		
17	00010001	•	
•			
•			
•		254	11111110
31	00011111	255	11111111

Figura 1-2 Tabla de conversión decimal a binario.

Ejercicio 1.1: *¿Cuál es el valor decimal de "11111100?*

Conversión decimal a binario

Inversamente, calculemos el equivalente binario de "11" en decimal:

$$\begin{array}{rl} 11 \div 2 = 5 \text{ resto } 1 \rightarrow 1 & \text{bit menos significativo (LSB)} \\ 5 \div 2 = 2 \text{ resto } 1 \rightarrow 1 & \\ 2 \div 2 = 1 \text{ resto } 0 \rightarrow 0 & \\ 1 \div 2 = 0 \text{ resto } 1 \rightarrow 1 & \text{bit más significativo (MSB)} \end{array}$$

El equivalente binario es 1011 (la lectura de la columna más a la derecha es de abajo arriba).
El equivalente binario de un número decimal se puede obtener por división sucesiva por 2 hasta que se obtenga un cociente 0.

Ejercicio 1.2: *¿Cuál es el equivalente binario de 257?*

Ejercicio 1.3: *Convertir 19 a binario y después, de nuevo, a decimal.*

Operación en binario

Las reglas aritméticas para números binarios son sencillas. Las reglas de la suma son:

$$\begin{array}{l} 0 + 0 = \quad 0 \\ 0 + 1 = \quad 1 \\ 1 + 0 = \quad 1 \\ 1 + 1 = (1)\ 0 \end{array}$$

donde (1) representa un "acarreo" de 1 (observe que "10" es el equivalente binario de "2" decimal). La resta binaria se efectuará "sumando el complemento" y se explicará cuando se aprenda cómo representar números negativos.

Ejemplo:

$$\begin{array}{r} (2) \\ + (1) \\ \hline = (3) \end{array} \qquad \begin{array}{r} 10 \\ + 01 \\ \hline 11 \end{array}$$

La suma se efectúa exactamente como en decimal, sumando las columnas de derecha a izquierda:

Suma de la columna más a la derecha:

$$\begin{array}{r} 10 \\ + 01 \\ \hline \end{array}$$

(0 + 1 = 1. Sin acarreo)

Suma de la siguiente columna:

```
  10
+ 01
----
  11    (1 + 0 = 1. Sin acarreo)
```

Ejercicio 1.4: *Calcular 5 + 10 en binario. Verificar que el resultado es 15.*

Algunos ejemplos adicionales de suma binaria:

```
  0010    (2)          0011    (3)
+ 0001    (1)        + 0001    (1)
--------------       --------------
= 0011    (3)        = 0100    (4)
```

Este último ejemplo ilustra la misión del acarreo.

Considerando los bits más a la derecha: 1 + 1 = (1) 0.
Se genera un acarreo de 1, que se debe sumar a los siguientes bits:

```
   001 — la columna 0 que se acaba de sumar
 + 000 —
 +   1   (acarreo)
-----------------
= (1) 0 — en donde (1) indica un nuevo acarreo
          en la columna 2.
```

El resultado final es: 0100.

Otro ejemplo:

```
  0111        (7)
+ 0011    +   (3)
-----------------
  1010    =  (10)
```

En este ejemplo se genera de nuevo un acarreo, hasta la columna más a la izquierda.

Ejercicio 1.5: *Calcular el resultado de:*

$$\begin{array}{r} 1111 \\ +\ 0001 \\ \hline =\ ? \end{array}$$

¿El resultado tiene 4 bits?

Con ocho bits es posible, por tanto, representar directamente los números "00000000" a "11111111", es decir, "0" a "255". Dos obstáculos aparecerán inmediatamente. Primero, sólo se representan números positivos. Segundo, la magnitud de estos números se limita a 255 si se emplean ocho bits solamente. Tratemos cada uno de estos problemas sucesivamente.

Representación binaria con signo

En una representación binaria con signo, el bit más a la izquierda se utiliza para indicar el signo del número. Tradicionalmente, "0" se emplea para señalar un número *positivo* mientras "1" se utiliza para señalar un número *negativo*. Según lo anterior, "11111111" representará −127, mientras que "01111111" representará +127. Ahora se pueden representar números positivos y negativos, pero se ha reducido la magnitud máxima a 127.

Ejemplo: "0000 0001" representa +1 (el "0" más a la izquierda es "+", seguido por "000 0001" = 1).

"1000 0001" es −1 (el dígito más a la izquierda "1" es "−").

Ejercicio 1.6: *¿Cuál es la representación de "−5" en binario con signo?*

Tratemos ahora el problema de la *magnitud*: con el fin de representar números grandes, será necesario utilizar un número más grande de bits. Por ejemplo, si se utilizan dieciséis bits (dos bytes) para la representación, se podrán representar números desde −32K a +32K en representación binaria con signo (en la jerga de ordenador, 1K significa 1024). El bit 15 se emplea para el signo, y los 15 bits restantes (bit 14 al bit 0) se utilizan para la magnitud: 2^{15} = 32K. Si esta magnitud es todavía demasiado pequeña, se emplearán 3 bytes o más. Si se desea representar enteros grandes, será necesario utilizar un gran número de bytes internamente para representarlos. Es por ello que la mayoría de los BASIC sencillos, y otros lenguajes, solamente proporcionan una precisión limitada para los enteros. De este modo, pueden emplear un formato interno más corto para los números que manipulan. Las mejores versiones de BASIC o de otros lenguajes proporcionan

un mayor número de cifras decimales significativas a costa de un mayor número de bytes para cada número.

Ahora solucionemos otro problema, el de la velocidad de cálculo. Vamos a tratar de realizar una suma en la representación binaria con signo que hemos introducido. Sea sumar "−5" y "+7".

+ 7 se representa por	00000111
− 5 se representa por	10000101
La suma binaria es	10001100, o −12

Este resultado no es el correcto, pues éste deberá ser +2. Con el fin de utilizar esta representación, se deben considerar acciones especiales que dependen del signo. Ello da lugar a una complejidad creciente y a un menor rendimiento. Dicho de otro modo, la suma binaria de números con signos no "funciona correctamente" Ello es fastidioso, pues es evidente que el ordenador no debe representar únicamente la información, sino también realizar operaciones aritméticas.

La solución de este problema la da la representación en *complemento a dos,* que se utilizará en lugar de la representación *binaria con signo.* Con el fin de introducir el complemento a dos, veremos, primeramente, una etapa intermedia: *el complemento a uno.*

Complemento a uno

En la representación en complemento a uno, todos los enteros positivos se representan en su formato binario correcto. Por ejemplo "+3" se suele representar por 00000011. Sin embargo, su opuesto "−3" se obtiene complementando cada bit en la representación original. Cada 0 se transforma en un 1 y cada 1 se transforma en un 0. En nuestro ejemplo, la representación en complemento a uno de "−3" será 11111100.

Otro ejemplo:

+2 es 00000010
−2 es 11111101

Observe que, en esta representación, los números positivos comienzan con un "0" a la izquierda, y los números negativos con un "1", también a la izquierda.

Ejercicio 1.7: *La representación de "+6" es "00000110". ¿Cuál es la representación de "−6" en complemento a uno?*

Como prueba, sumemos -4 y $+6$:

	-4 es 11111011		
	$+6$ es 00000110		
la suma es:	(1) 00000001	en donde (1) indica un acarreo.	

El "resultado correcto" será "2", o "00000010".

Probemos de nuevo:

	-3 es 11111100
	-2 es 11111101
la suma es:	(1) 00000001

o sea "1" más un acarreo. El resultado correcto debe ser "-5". La representación de "-5" es 11111010. La suma no se efectúa correctamente.

Esta representación permite positivos y negativos. Sin embargo, el resultado de una suma ordinaria no siempre es correcto. Todavía se empleará otra representación. Se deduce del complemento a uno y se llama la representación en complemento a dos.

Representación en complemento a dos

En la representación en complemento a dos, los números positivos se representan también, habitualmente, en binario con signo, igual que en complemento a uno. La diferencia radica en la representación de *números negativos.* La representación de un número negativo en complemento a dos se obtiene calculando primeramente el complemento a uno y luego *sumándole uno.* Veámoslo con un ejemplo:

$+3$ se representa en binario con signo por 00000011. Su representación en complemento a uno es 11111100. El complemento a dos se obtiene añadiendo uno, esto es, 11111101.

Intentemos realizar una suma:

(3)	00000011
+ (5)	+ 00000101
= (8)	= 00001000

El resultado es correcto.

Realicemos una resta:

(3)	00000011
(−5)	+ 11111011
	= 11111110

Identifiquemos el resultado calculando su complemento a dos:

el complemento a uno de 11111110 es	00000001
Sumando 1	+ 1
por tanto, el complemento a dos es	00000010 o +2

Nuestro resultado anterior, "11111110" representa "−2". El resultado es correcto.

Hemos probado la suma y la resta, y el resultado fue correcto (ignorando el acarreo). Parece que el complemento a dos funciona.

Ejercicio 1.8: *¿Cuál es la representación de "+127" en complemento a dos?*

Ejercicio 1.9: *¿Cuál es la representación de "−128" en complemento a dos?*

Sumemos ahora +4 y −3 (la resta se realiza sumando el complemento a dos):

	+4 es	00000100
	−3 es	11111101
El resultado es:	(1)	00000001

Si se ignora el acarreo, el resultado es 00000001, esto es, "1" en decimal, que es el resultado correcto. Sin dar la demostración matemática completa, indiquemos simplemente que esta representación sí funciona. En complemento a dos, es posible sumar o restar números con signo con independencia del signo. Utilizando las reglas habituales de la suma binaria, se obtiene el resultado correctamente, incluyendo el signo. Se ignora el acarreo y ello significa una ventaja considerable. Si no fuera así, sería preciso corregir el signo del resultado cada vez, originando un tiempo de suma o resta más lento.

Con el fin de ser exactos, indiquemos que el complemento a dos es simplemente la representación más conveniente para utilizar por los procesadores más sencillos, tales como los microprocesadores. En los procesadores complejos, se pueden utilizar otras representaciones. Por ejemplo, se puede utilizar un complemento a uno, pero requiere circuitería especial para "corregir el resultado".

Se supondrá, a partir de este momento, que todos los enteros con signo se representarán internamente en notación de complemento a dos. Ver en la figura 1-3 una tabla de complemento a dos de los números.

Ejercicio 1.10: *¿Cuáles son los números más grande y más pequeño que se pueden representar en notación de complemento a dos con un solo byte?*

Ejercicio 1.11: *Calcule el complemento a dos de 20. Después, calcule el complemento a dos de su resultado. ¿Obtiene de nuevo 20?*

Los ejemplos siguientes servirán para demostrar las reglas del complemento a dos. En particular, C denota una condición de posible acarreo [o *acarreo negativo* ("borrow")]. (Es el bit 8 del resultado).

V indica un *desbordamiento* ("overflow") en complemento a dos, es decir, cuando el signo del resultado se cambia "accidentalmente" ya que los números son demasiado grandes. Es, esencialmente, un acarreo interno del bit 6 al bit 7 (bit de signo). Se aclarará más adelante.

Explicaremos ahora el papel del acarreo "C" y el desbordamiento "V".

El acarreo C

He aquí un ejemplo de un acarreo:

```
    (128)           10000000
   +(129)          +10000001
  ----------------------------
    (257) = (1)     00000001
```

en donde (1) indica un acarreo.

El resultado requiere un noveno bit (bit "8", ya que el bit más a la derecha es "0"). Es el bit de acarreo.

Si suponemos que el acarreo es el noveno bit del resultado, interpretamos el resultado como 100000001 = 257.

Sin embargo, el acarreo se debe identificar y manipular con cuidado. En el interior del microprocesador, los registros utilizados para guardar la

+	Código en complemento a dos	−	Código en complemento a dos
+ 127	01111111	− 128	10000000
+ 126	01111110	− 127	10000001
+ 125	01111101	− 126	10000010
. . .		− 125	10000011
		. . .	
+ 65	01000001	− 65	10111111
+ 64	01000000	− 64	11000000
+ 63	00111111	− 63	11000001
. . .		. . .	
+ 33	00100001	− 33	11011111
+ 32	00100000	− 32	11100000
+ 31	00011111	− 31	11100001
. . .		. . .	
+ 17	00010001	− 17	11101111
+ 16	00010000	− 16	11110000
+ 15	00001111	− 15	11110001
+ 14	00001110	− 14	11110010
+ 13	00001101	− 13	11110011
+ 12	00001100	− 12	11110100
+ 11	00001011	− 11	11110101
+ 10	00001010	− 10	11110110
+ 9	00001001	− 9	11110111
+ 8	00001000	− 8	11111000
+ 7	00000111	− 7	11111001
+ 6	00000110	− 6	11111010
+ 5	00000101	− 5	11111011
+ 4	00000100	− 4	11111100
+ 3	00000011	− 3	11111101
+ 2	00000010	− 2	11111110
+ 1	00000001	− 1	11111111
+ 0	00000000		

Figura 1-3 Tabla de complemento a dos.

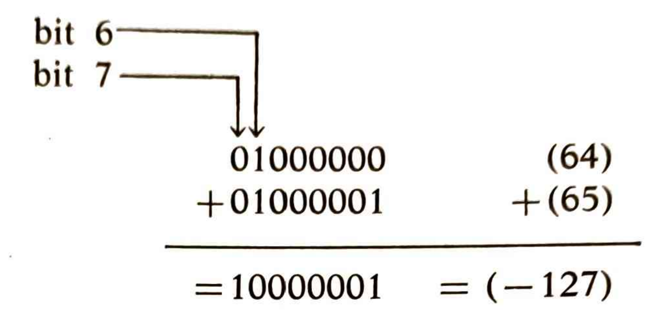
información suelen ser de ocho bits. Cuando se almacena el resultado, solamente se conservarán los bits 0 a 7.

En consecuencia, un acarreo requiere siempre una acción especial: se debe detectar por instrucciones especiales, que luego se procesan. El proceso

del acarreo significa almacenarlo en alguna parte (con una instrucción especial) o ignorarlo, o bien decidir que se trata de un error (si el resultado más grande permitido es "11111111").

Desbordamiento V

He aquí un ejemplo de desbordamiento:



Un acarreo interno se ha generado del bit 6 al bit 7. Ello se denomina un *desbordamiento* (overflow).

El resultado es ahora negativo, "por accidente". Esta situación debe ser detectada, de modo que se pueda corregir.

Examinemos otra situación:

```
         11111111    (−1)
        +11111111   +(−1)
-----------------------------
=(1)     11111110   =(−2)
  ↓
acarreo
```

En este caso, un acarreo interno se ha generado del bit 6 al bit 7, y también del bit 7 al bit 8 (el "acarreo" verdadero C que hemos visto en la sección anterior). Las reglas de la aritmética en complemento a dos especifican que se debe ignorar este acarreo. El resultado es entonces correcto.

Lo que se debe a que el acarreo del bit 6 al bit 7 no cambia el bit de signo.

No es una condición de *desbordamiento*. Cuando se opera con números negativos, el desbordamiento no es simplemente un acarreo del bit 6 al bit 7. Examinemos otro ejemplo:

```
         11000000     (−64)
       + 10111111     (−65)
-------------------------------
=(1)     01111111     (+127)
  ↓
acarreo
```

Esta vez, no ha sido un acarreo interno del bit 6 al bit 7, sino que ha sido un acarreo externo. El resultado es incorrecto, pues el bit 7 se ha modificado. Se debe indicar una condición de desbordamiento.

El desbordamiento se producirá en cuatro casos:

1 — Suma de números positivos grandes.
2 — Suma de números negativos grandes.
3 — Resta de un número positivo grande de un número negativo grande.
4 — Resta de un número negativo grande de un número positivo grande.

Perfeccionemos ahora nuestra definición del desbordamiento.

Técnicamente, el indicador de desbordamiento, un bit especial reservado para este fin y que se denomina "indicador" (flag), será puesto a 1 cuando hay un acarreo del bit 6 al bit 7 y ningún acarreo externo, o bien cuando no hay acarreo del bit 6 al bit 7 pero hay un acarreo externo. Esto indica que el bit 7, es decir, el signo del resultado, se ha modificado accidentalmente. Para el lector interesado por la parte técnica, el indicador de desbordamiento se establece realizando la función OR exclusiva (O exclusiva) del acarreo de entrada al bit 7 y del acarreo de salida del mismo (el bit de signo). Prácticamente, todos los microprocesadores disponen de un indicador de desbordamiento especial que detecta esta condición, la cual requiere una acción correctora.

El desbordamiento significa que el resultado de una suma o resta requiere más bits que los disponibles en el registro estándar de ocho bits utilizado para contener el resultado.

El acarreo y el desbordamiento

Los bits de acarreo y desbordamiento se llaman "indicadores de estado" (flags). Están presentes en todo microprocesador y en el siguiente capítulo se aprenderá a utilizarlos para programación efectiva. Estos dos indicadores están situados en un registro especial llamado "registro de estado". Este registro contiene también otros indicadores adicionales cuya función se explicará en el capítulo 4.

Ejemplos

Ilustremos ahora la operación del acarreo y el desbordamiento en ejemplos concretos. En cada ejemplo, el símbolo V significa el desbordamiento y C el acarreo.

Si no ha habido ningún desbordamiento V = 0. Si lo ha habido, V = 1 (igual para el acarreo C). Recuérdese que según las reglas de la aritmética en complemento a dos, se ignora el acarreo (no se da aquí la demostración matemática).

Positivo + Positivo

	00000110	(+6)		
+	00001000	(+8)		
=	00001110	(+14)	V:0	C:0

(CORRECTO)

Positivo + Positivo con desbordamiento

	01111111	(+127)		
+	00000001	(+1)		
=	10000000	(−128)	V:1	C:0

Lo anterior no es válido ya que se ha producido un desbordamiento. (ERROR)

Positivo + Negativo (resultado positivo)

	00000100	(+4)		
+	11111110	(−2)		
=(1)	00000010	(+2)	V:0	C:1 (despreciable)

(CORRECTO)

Positivo + Negativo (resultado negativo)

	00000010	(+2)		
+	11111100	(−4)		
=	11111110	(−2)	V:0	C:0

(CORRECTO)

Negativo + Negativo

```
        11111110    (−2)
+       11111010    (−4)
----------------------------
=(1)    11111010    (−6)      V:0      C:1 (despreciable)
```

(CORRECTO)

Negativo + Negativo con desbordamiento

```
        10000001    (−127)
+       11000010    (−62)
----------------------------
=(1)    01000011    (67)      V:1      C:1
```

(ERROR)

Esta vez se ha producido un *"desbordamiento negativo"* ("underflow"), al sumar dos números negativos grandes. El resultado será −189, que es demasiado grande para representarse por ocho bits.

Ejercicio 1.12: *Completar las siguientes sumas. Indique el resultado, el acarreo C, el desbordamiento V, y si el resultado es correcto o no:*

```
 10111111   (......)              11111010   (......)
+11000001   (......)             +11111001   (......)
---------                        ---------
=...........  V:......  C:......    =...........  V:......  C:......
□ CORRECTO      □ ERROR            □ CORRECTO      □ ERROR

 00010000   (......)              01111110   (......)
+01000000   (......)             +00101010   (......)
---------                        ---------
=...........  V:......  C:......    =...........  V:......  C:......
□ CORRECTO      □ ERROR            □ CORRECTO      □ ERROR
```

Ejercicio 1.13: *¿Puede indicar un ejemplo de desbordamiento obtenido al sumar un número positivo y un número negativo? ¿Por qué?*

Representación en formato fijo

Sabemos ahora representar enteros con signo. Sin embargo, no hemos

solucionado el problema de la magnitud. Si queremos representar enteros más grandes, necesitaremos varios bytes. Con el fin de ejecutar eficazmente operaciones aritméticas, es necesario utilizar un número fijo de bytes en vez de variable. En consecuencia, una vez que se haya elegido el número de bytes, la máxima magnitud del número que se puede representar es fija.

Ejercicio 1.14: *¿Cuáles son los números más grande y más pequeño que se pueden representar en complemento a dos con dos bytes?*

El problema de la magnitud de los números

Cuando sumamos números nos hemos limitado a ocho bits porque el procesador que utilizaremos opera internamente sólo con ocho bits a la vez. Sin embargo, ello nos limita a números comprendidos entre -128 y $+127$. Lógicamente, eso no es suficiente para muchas aplicaciones.

Se utilizará precisión múltiple para incrementar el número de dígitos que se pueden representar. Se puede utilizar, en tal caso, un formato de dos, tres o N bytes. Examinemos, por ejemplo, el formato de "doble precisión" de 16 bits:

00000000	00000000	es "0"
00000000	00000001	es "1"
01111111	11111111	es "32767"
11111111	11111111	es "−1"
11111111	11111110	es "−2"

Ejercicio 1.15: *¿Cuál es el entero negativo más grande que se puede representar en complemento a dos en formato de triple precisión?*

No obstante, este método presenta inconvenientes. Cuando se suman dos números, por ejemplo, se tendrán que sumar generalmente los ocho bits a la vez. Esto se explicará en el capítulo 3 (Técnicas de programación básicas). Resulta por ello un proceso más lento. También esta representación utiliza 16 bits para cualquier número, aun en el caso de que se pueda representar con sólo ocho bits. Es normal, por tanto, utilizar 16 o quizás 32, pero raras veces más.

Consideremos el siguiente punto importante: cualquiera que sea el número de bits, N, elegido para la representación en complemento a dos, es fijo. Si cualquier resultado, o cálculo intermedio, genera un número que requiera más de N bits, se perderán algunos bits. El programa suele conservar

los N bits más a la izquierda (los más significativos) y elimina los de orden más bajo. Ello se denomina "truncamiento del resultado".

He aquí un ejemplo en el sistema decimal, utilizando una representación de seis cifras:

$$
\begin{array}{r}
123456 \\
\times \quad 1.2 \\
\hline
246912 \\
123456 \\
\hline
= 148147.2
\end{array}
$$

El resultado exige 7 dígitos. El "2" después del punto decimal se despreciará y el resultado final será 148147. Se ha truncado. Habitualmente, mientras no se pierda la posición del punto decimal, este método se utiliza para ampliar la gama de las operaciones posibles, a expensas de la precisión.

El problema es igual en binario. Los detalles de una multiplicación binaria se darán en el capítulo 3.

Esta representación de formato fijo puede producir una pérdida de precisión, pero puede ser suficiente para cálculos usuales u operaciones matemáticas.

Lamentablemente, en el caso de contabilidad, no es admisible ninguna pérdida de precisión. Por ejemplo, si un cliente introduce en una caja registradora un total grande, no sería aceptable tener que pagar una cantidad de cinco cifras, que se aproximaría a la unidad monetaria básica. Es preciso utilizar otra representación donde sea esencial la precisión en el resultado. La solución normalmente utilizada es *BCD*, o *decimal codificado en binario.*

El código BCD

El principio utilizado en la representación de números en BCD es codificar cada cifra decimal por separado, y utilizar tantos bits como sean necesarios para representar exactamente el número completo. Con el fin de codificar cada uno de los dígitos de 0 a 9, se necesitan cuatro bits. Tres bits solamente darán ocho combinaciones y, en consecuencia, no pueden codificar las diez cifras. Cuatro bits permiten dieciséis combinaciones y son, por tanto, suficientes para codificar las cifras "0" a "9". Se puede observar, también, que seis de los posibles códigos no se utilizarán en la representación BCD (fig. 1-4). Esto redundará, posteriormente, en un posible problema durante las sumas y restas, que se tendrá que solucionar. Ya que solamente se necesitan cuatro bits para codificar un dígito BCD, en cada byte se pueden codificar dos dígitos BCD. Esto se llama "*BCD compacto*".

Código	Símbolo BCD	Código	Símbolo BCD
0000	0	1000	8
0001	1	1001	9
0010	2	1010	no utilizado
0011	3	1011	no utilizado
0100	4	1100	no utilizado
0101	5	1101	no utilizado
0110	6	1110	no utilizado
0111	7	1111	no utilizado

Figura 1-4 Tabla BCD.

Por ejemplo, "00000000" será "00" en BCD. "10011001" será "99".

Un código BCD se lee de la forma siguiente:

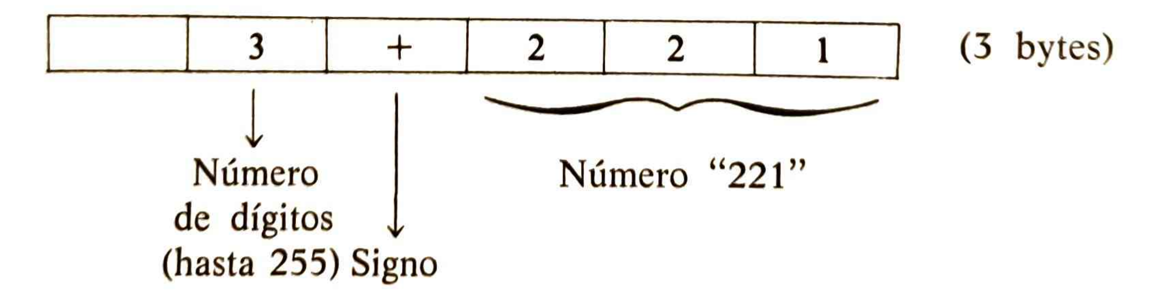
dígito "2" en BCD ←— 0010 0001
dígito "1" en BCD ←——————┘
número 21 en BCD

Ejercicio 1.16: *¿Cuáles son las representaciones en BCD de "29" y de "91"?*

Ejercicio 1.17: *¿Es "10100000" una representación BCD correcta? ¿Por qué?*

Se utilizarán tantos bytes como sean necesarios para representar todos los dígitos BCD. Generalmente, se utilizarán uno o más *nibbles* (cuaterna), o sea, el número total de dígitos utilizado. Otro nibble, o byte, se utilizará para indicar la posición del punto decimal. No obstante, pueden variar el convenio.

He aquí un ejemplo de una representación BCD de enteros de varios bytes:



Esto representa +221

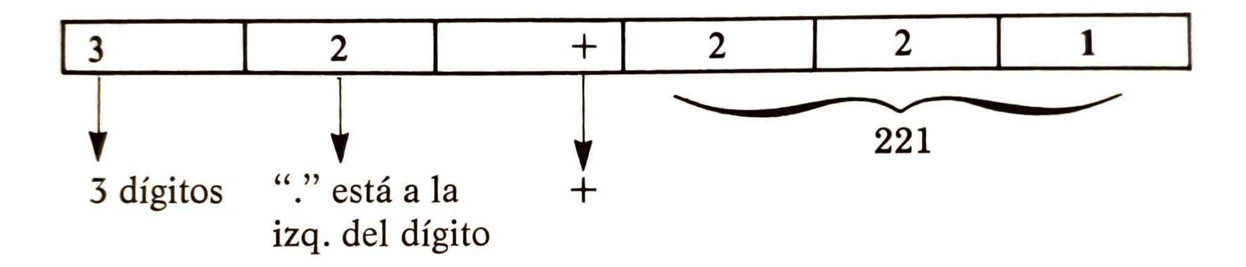
(El signo, por ejemplo, se puede representar por 0000 para + y 0001 para −).

Ejercicio 1.18: *Utilizando el mismo convenio, represente "−23123". Indíquelo en formato BCD, como anteriormente, y después en binario.*

Ejercicio 1.19: *Indique la representación BCD de "222" y "111" y, después, el resultado de 222 × 111. (Calcule el resultado manualmente y luego indíquelo en la representación anterior).*

La representación BCD puede aplicarse fácilmente a números decimales.

Por ejemplo, +2.21 se puede representar por:



La ventaja de BCD es que proporciona resultados absolutamente correctos. El inconveniente es que utiliza una gran cantidad de memoria y redunda en operaciones aritméticas lentas. Esto es aceptable solamente en un ámbito de contabilidad y no se suele utilizar en otros casos.

Ejercicio 1.20: *¿Cuántos bits son necesarios para codificar "9999" en BCD? ¿Y en complemento a dos?*

Hemos solucionado los problemas asociados con la representación de enteros, enteros con signo e incluso enteros grandes. Incluso hemos presentado un método posible de representación de números decimales en notación BCD. Examinemos ahora el problema de representación de números decimales en un formato de longitud fija.

Representación de coma (punto) flotante

El principio básico es que los números decimales deben ser representados con un formato fijo. Para no malgastar bits, la representación *normaliza* todos los números.

Por ejemplo, "0.000123" malgasta tres ceros a la izquierda del número, que no tienen significado excepto para indicar la posición del punto decimal.

La normalización de este número lo transformará en 0.123×10^{-3}. "123" se denomina *mantisa normalizada* y "-3" es el *exponente*. Se ha normalizado este número eliminando todos los ceros no significativos a la izquierda del mismo y ajustando el exponente.

Consideremos otro ejemplo:

22.1 se normaliza como 0.221×10^2

o $M \times 10^E$ donde M es la mantisa y E es el exponente.

Se puede ver fácilmente que un número normalizado se caracteriza por una mantisa menor que 1 y mayor o igual a 0.1, en todos los casos en que el número no sea cero. Dicho de otro modo, ello se puede representar matemáticamente por:

$$0.1 \leqslant M < 1 \quad \text{o} \quad 10^{-1} \leqslant M < 10^0$$

De forma análoga, en la representación binaria:

$$2^{-1} \leqslant M < 2^0 \quad (\text{o } 0.5 \leqslant M < 1)$$

En donde M es el valor absoluto de la mantisa (despreciando el signo).

Por ejemplo:

111.01 se normaliza como: 0.11101×2^3.

La mantisa es 11101.

El exponente es 3.

Ahora que hemos definido el principio de la representación, examinemos el formato real. Una representación típica de coma flotante aparece a continuación.

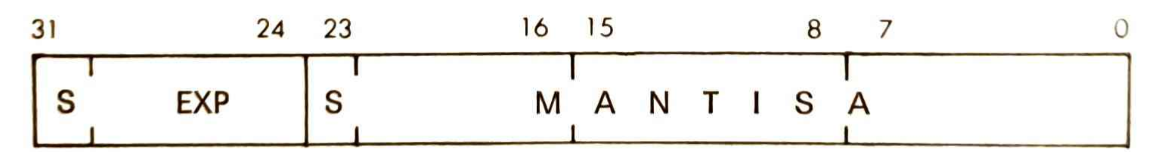


Figura 1-5 Representación típica de coma flotante.

En la representación utilizada en este ejemplo, se usan cuatro bytes o sea un total de 32 bits. El primer byte de la izquierda de la ilustración se utiliza para representar el exponente. Tanto el exponente como la mantisa se re-

presentarán en complemento a dos. Como resultado, el máximo exponente será −128. "S" en la figura 1-5 significa el bit de signo.

Se utilizan tres bytes para representar la mantisa. Como el primer bit en la representación de complemento a dos indica el signo, ello deja 23 bits para la representación de la magnitud de la mantisa.

Ejercicio 1.21: *¿Cuántos dígitos decimales puede representar la mantisa con los 23 bits?*

Se trata de un ejemplo de representación de coma flotante. Es posible emplear solamente tres bytes o utilizar más. La representación de cuatro bytes anteriormente propuesta representa una solución de compromiso razonable en términos de exactitud, magnitud de números, utilización de almacenamiento y eficacia en operación aritmética.

Ya hemos abordado los problemas inherentes a la representación de números y sabemos cómo representarlos en forma entera, con un signo o en forma decimal. Examinemos, ahora, cómo representar internamente datos alfanuméricos.

Representación de datos alfanuméricos

La representación de datos alfanuméricos (esto es, caracteres) es muy sencilla: todos los caracteres se codifican en un código de ocho bits. Sólo dos códigos son de uso general en el sector de los ordenadores: el código ASCII y el código EBCDIC. ASCII es la abreviatura de "American Standard Code for Information Interchange" (Código Normalizado Americano para Intercambio de Información) y se utiliza universalmente en el campo de los microprocesadores. EBCDIC es una variante de ASCII utilizada por IBM y, por consiguiente, no se emplea en el campo de los microordenadores a no ser que haya una interconexión con un terminal de IBM.

Examinemos sucintamente la codificación en ASCII. Tenemos que codificar 26 letras del alfabeto para mayúsculas y minúsculas, más 10 símbolos numéricos y, eventualmente, 20 símbolos especiales adicionales. Ello puede realizarse fácilmente con 7 bits, que permiten 128 códigos posibles (fig. 1-6). Todos los caracteres se codifican, pues, en 7 bits. El octavo bit, cuando se utiliza, es el de *paridad*. La paridad es una técnica para comprobar que el contenido de un byte no se ha cambiado accidentalmente. El número de "1" en el byte es objeto de contaje y el octavo bit se pone a "1" si el contaje fue impar, con lo que se obtiene un total par. Ello se denomina paridad par. También se puede emplear la paridad impar, que consiste en escribir el

octavo bit (el más a la izquierda) de modo que sea impar el número total de "1" en el byte.

Ejemplo: Calculemos el bit de paridad para "0010011" con el empleo del método de la paridad par. El número de unos (1) es 3. El bit de paridad debe ser, pues, un 1 de modo que el número total de bits sea 4, esto es, par. El resultado es 10010011, en donde el 1 a la izquierda es el bit de paridad y 0010011 identifica el carácter.

La tabla de códigos ASCII de 7 bits se indica en la figura 1-6. En la práctica se utiliza "tal como está" (esto es, sin paridad), añadiendo un 0 en la posición más a la izquierda, o bien con paridad, añadiendo el bit adicional adecuado a la izquierda.

Ejercicio 1.22: *Calcular la representación de 8 bits de los dígitos "0" a "9"*

NÚMEROS DE BIT								0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_7	b_6	b_5	b_4	b_3	b_2	b_1	HEX 1 / HEX 0	0	1	2	3	4	5	6	7
			0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
			0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
			0	0	1	0	2	STX	DC2	"	2	B	R	b	r
			0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
			0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
			0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
			0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
			0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
			1	0	0	0	8	BS	CAN	(	8	H	X	h	x
			1	0	0	1	9	HT	EM	)	9	I	Y	i	y
			1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
			1	0	1	1	11	VT	ESC	+	;	K	[	k	{
			1	1	0	0	12	FF	FS	,	<	L	\	l	|
			1	1	0	1	13	CR	GS	-	=	M	]	m	}
			1	1	1	0	14	SO	RS	.	>	N	^	n	~
			1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Figura 1-6 Tabla de conversión ASCII.

con el empleo de la paridad par. (Este código se utilizará en ejemplos de aplicación del capítulo 8.)

Ejercicio 1.23: *Lo mismo para las letras "A" a "F".*

Ejercicio 1.24: *Con el empleo del código ASCII sin paridad (en donde es "0" el bit más a la izquierda), indicar el contenido binario de los 4 bytes siguientes:*

"A", "T", "S", "X"

En campos especializados tales como el de las telecomunicaciones, pueden emplearse otras codificaciones tales como códigos de corrección de errores, que caen fuera del alcance de este libro.

Hemos examinado las representaciones habituales para programas y datos en el interior del ordenador. Examinemos ahora las posibles representaciones externas.

REPRESENTACIÓN EXTERNA DE INFORMACIÓN

La representación externa se refiere a la forma en que la información se presenta al *usuario,* que suele ser el programador. La información puede presentarse externamente en prácticamente tres formatos: binario, octal o hexadecimal y simbólico.

1. Binario

Se ha visto que la información se almacena internamente en *bytes* u *octetos,* que son secuencias de ocho *bits* (ceros o unos). A veces es deseable visualizar esta información interna directamente en su formato binario y se denomina *representación binaria.* Un ejemplo simple lo proporcionan los diodos emisores de luz (LED) que, fundamentalmente, son luces miniatura en el panel frontal del microordenador. En el caso de un microprocesador de 8 bits, un panel frontal estará provisto de ocho LED para visualizar el contenido de cualquier registro interno (se utiliza un registro para retener ocho bits de información y se describirá en el capítulo 2). Un LED iluminado indica un "1". Un cero viene indicado por un LED que no se ha iluminado. Dicha representación binaria puede utilizarse para la depuración precisa de un programa complejo, sobre todo si implica entrada/salida, pero, naturalmente, no es práctico a nivel humano. Esta es la razón por la que, en la mayor parte de los casos, se desea examinar la información en forma simbólica. Así, "9" es mucho más fácil de comprender, o de recordar, que

"1001". Se han concebido representaciones más adecuadas que mejoran la interconexión persona-máquina.

2. *Octal y hexadecimal*

Con estas representaciones "octal" y "hexadecimal" se codifican respectivamente tres y cuatro bits binarios en un símbolo singular. En el sistema octal, cualquier combinación de tres bits binarios se representa por un número comprendido entre 0 y 7.

En octal se tiene un formato con tres bits, en donde cada combinación de tres bits se representa por un símbolo entre 0 y 7:

Binario	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Figura 1-7 Símbolos del código octal.

Por ejemplo, el número binario "00 100 100" se representa por:

$$\begin{matrix} 00 & 100 & 100 \\ \downarrow & \downarrow & \downarrow \\ 0 & 4 & 4 \end{matrix}$$

o "044" en octal.

Otro ejemplo: 11 111 111 es:

$$\begin{matrix} 11 & 111 & 111 \\ \downarrow & \downarrow & \downarrow \\ 3 & 7 & 7 \end{matrix}$$

o "377" en octal.

Inversamente, el octal "211" representa:

010 001 001

o "10001001" en binario.

La representación en octal se ha empleado tradicionalmente en los antiguos ordenadores, en los que se empleaban diversos números de bits desde 8 a quizás 64. En los últimos años, con el predominio de los microprocesadores de ocho bits, ha llegado a normalizarse el formato de ocho bits y se utiliza otra representación más práctica, que es la *hexadecimal*.

En la representación hexadecimal, un grupo de cuatro bits se codifica como un dígito hexadecimal. Los dígitos hexadecimales se representan por los símbolos del 0 al 9 y por las letras A, B, C, D, E, F. Por ejemplo, "0000" se representa por "0", "0001" se representa por "1" y "1111" se representa por la letra "F" (fig. 1-8).

Ejemplo: $\underbrace{1010}_{A}\ \underbrace{0001}_{1}$ en binario se representa por A 1 en hexadecimal.

Ejercicio 1.25: *¿Cuál es la representación hexadecimal de "10101010"?*

Ejercicio 1.26: *¿Cuál es el equivalente binario de "FA" en hexadecimal?*

Ejercicio 1.27: *¿Cuál es el equivalente octal de "01000001"?*

La representación hexadecimal ofrece la ventaja de codificar ocho bits en solamente dos dígitos, lo que es más fácil de visualizar o de memorizar y más rápido de teclear en un ordenador que su equivalente binario. Por consiguiente, en la mayoría de los microordenadores modernos, el hexadecimal es el método preferido de representación para grupos de bits.

Naturalmente, siempre que la información existente en la memoria tenga un significado, tal como la representación de textos o de números, el código hexadecimal no es adecuado para representar el significado de dicha información cuando haya de emplearse por personas.

Representación simbólica

La representación simbólica se refiere a la representación externa de la información en forma simbólica real. Por ejemplo, los números decimales se representan como tales números decimales y no como secuencias de bits o de símbolos hexadecimales. Análogamente, el texto se representa como tal. Naturalmente, la representación simbólica es más práctica para el usuario. Se utiliza siempre que se disponga de un dispositivo visualizador adecuado, tal como un tubo de rayos catódicos (TRC) o una impresora (la visualización en TRC está constituida por una pantalla de tipo de televisión utilizada para visualizar textos o gráficos). Lamentablemente, en sistemas más pequeños

DECIMAL	BINARIO	HEXADECIMAL	OCTAL
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Figura 1-8 Códigos hexadecimales.

tales como microordenadores de una sola placa, no resulta económico el empleo de dichos dispositivos de visualización y el usuario se ve obligado a la comunicación en hexadecimal con el ordenador.

Resumen de las representaciones externas

La representación simbólica de la información es la más deseable puesto que es la más natural para el usuario humano. Sin embargo, exige un sistema de interconexión caro en la forma de un teclado alfanumérico, más una impresora o una visualización de TRC. Por este motivo, no puede disponerse

de la misma en los sistemas menos caros. Entonces se emplea un tipo alternativo de representación y, en este caso, la predominante es la hexadecimal. Solamente en casos no frecuentes relacionados con la depuración al nivel de software, o hardware, se utiliza la representación binaria. En *binario* se utiliza directamente el contenido de registros de memoria en formato binario. (La utilidad de una visualización binaria directa en un panel frontal siempre fue tema de fuerte controversia emocional y no vamos a entrar en ella en el marco de este libro).

Hemos visto cómo representar la información de forma interna y externa. Examinaremos, a continuación, el mismo microprocesador real que manipulará esta información.

Ejercicios adicionales

Ejercicio 1.28: *¿Cuál es la ventaja del complemento a dos sobre las demás representaciones utilizadas para representar números con signo?*

Ejercicio 1.29: *¿Cómo representaría "1024" en binario directo? ¿Y en binario con signo? ¿Y en complemento a dos?*

Ejercicio 1.30: *¿Qué es el bit V? ¿Debe comprobarlo el programador después de una adición o de una sustracción?*

Ejercicio 1.31: *Calcular el complemento a dos de "+16", "+17", "+18", "−16", "−17", "−18".*

Ejercicio 1.32: *Indicar la representación hexadecimal del siguiente texto, que se ha almacenado internamente en formato ASCII sin ninguna paridad: ="MESSAGE".*

2 Organización del hardware del 6502

INTRODUCCIÓN

Para programar a un nivel elemental, no es necesario comprender en detalle la estructura interna del procesador que se está utilizando. Sin embargo, para programar de modo más eficaz, tal comprensión es necesaria. El objetivo de este capítulo es presentar los conceptos fundamentales de hardware necesarios para comprender el funcionamiento del sistema 6502. El sistema completo de microordenador incluye no solamente el microprocesador (en este caso el 6502), sino también otros componentes. Este capítulo presenta el 6502 propiamente dicho, mientras que los otros dispositivos (principalmente de entrada/salida) se presentarán en el capítulo 7.

Revisaremos aquí la arquitectura básica de un sistema de microordenador y después estudiaremos más detalladamente la organización interna del 6502. Examinaremos, en particular, los diversos registros y más adelante estudiaremos la ejecución del programa y el mecanismo de control secuencial. Desde el punto de vista del hardware, este capítulo es solamente una presentación simplificada. Para más detalles puede consultarse el libro "Del chip al sistema" del mismo autor, editado en castellano por Marcombo, S. A.

ARQUITECTURA DEL SISTEMA

La arquitectura del sistema de microordenador aparece en la figura 2-1. La unidad del microprocesador (MPU), que aquí será un 6502, aparece a la izquierda de la ilustración. Realiza las funciones de una *unidad central de proceso* (CPU) en una sola pastilla (chip) e incluye una *unidad aritmética*

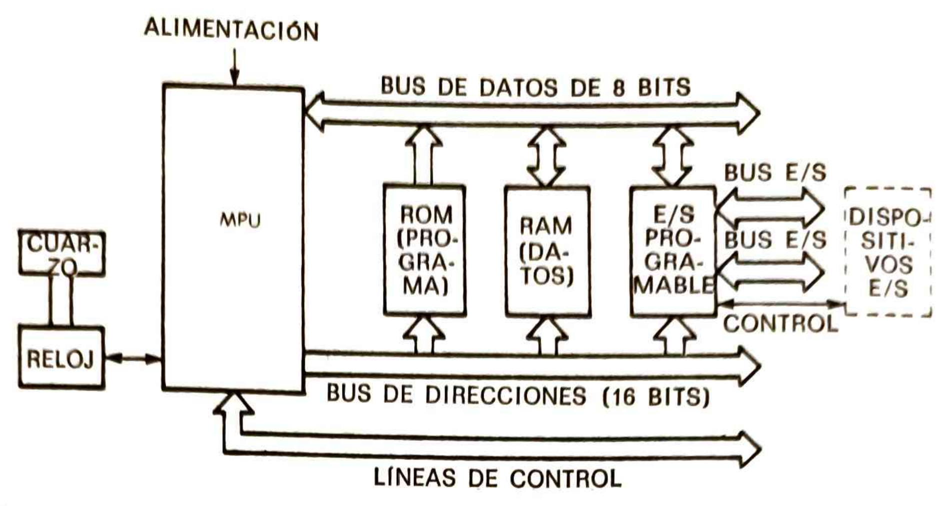


Figura 2-1 Arquitectura de un sistema con microprocesador estándar.

y lógica (ALU), con sus registros internos y una *unidad de control* (CU) encargada de secuenciar el sistema.

La MPU crea tres *buses:* un *bus de datos* de 8 bits bidireccional, que aparece en la parte superior de la ilustración, un *bus de direcciones* de 16 bits unidireccional y un *bus de control* que aparece en la parte inferior de la ilustración. Describamos la función de cada uno de los tres *buses.*

El *bus de datos* transporta *los datos* que se intercambian por los diversos elementos del sistema. Generalmente, se transportarán datos desde la memoria a la MPU, desde la MPU a la memoria, o desde la MPU a una pastilla de entrada/salida. (Una pastilla de entrada/salida es un componente encargado de comunicar con un dispositivo externo.)

El *bus de direcciones* transporta una *dirección* generada por la MPU, que seleccionará un registro interno en una de las pastillas conectadas al sistema. Esta dirección indica la fuente, o el destino, de los datos que se transmiten por el bus de datos.

El *bus de control* transporta las diferentes señales de sincronización requeridas por el sistema.

Habiendo descrito la misión de estos buses, conectemos ahora los componentes adicionales requeridos por un sistema completo.

Cada MPU exige una referencia de tiempo precisa, que es proporcionada por un *reloj* y un *cristal.* En la mayoría de los microprocesadores "antiguos", el oscilador de reloj es exterior a la MPU y requiere una pastilla adicional. En la mayoría de los microprocesadores recientes, el oscilador de reloj se suele incorporar a la MPU. Sin embargo, el cristal de cuarzo es siempre exterior al sistema debido a su volumen. El cristal y el reloj aparecen a la izquierda del bloque MPU en la ilustración.

Centremos nuestra atención en los otros elementos del sistema. De izquierda a derecha se distinguen en la ilustración:

La *ROM* que es la *memoria de sólo lectura* y contiene el *programa* del sistema. La ventaja de la ROM es que su contenido es permanente y no desaparece cuando el sistema se desconecta. La ROM, por tanto, contiene siempre un programa *cargador* (bootstrap-carga inicial) o un *monitor* (sus funciones se explicarán más adelante) para permitir el funcionamiento inicial del sistema. En un contexto de control de proceso, casi todos los programas residirán en ROM, de modo que nunca se cambiarán probablemente. En tal caso, el usuario industrial tiene que proteger el sistema contra fallos de alimentación: los programas no pueden ser volátiles. Deben estar en ROM.

Sin embargo, en un ámbito de aficiones personales o en el desarrollo de un programa (cuando el programador comprueba el programa), la mayoría de los programas residirán en RAM de modo que se puedan cambiar fácilmente. Posteriormente, pueden permanecer en RAM o ser transferidos a ROM, si se desea. Sin embargo, la RAM es volátil. Su contenido se pierde cuando se desconecta la alimentación.

La *RAM (memoria de acceso aleatorio)* es la memoria de lectura/escritura del sistema. En el caso de un sistema de control, la magnitud de la RAM será generalmente pequeña (solamente para datos). Por el contrario, en un contexto de desarrollo de programa, la magnitud de RAM será grande, de modo que contendrá programas más software de desarrollo. Todo el contenido de RAM se debe cargar desde un dispositivo externo, antes de su uso.

Finalmente, el sistema contendrá una o más pastillas de interface, de modo que se pueda comunicar con el mundo exterior. La pastilla de interface más frecuentemente utilizada es el *"PIO"* o pastilla de entrada-salida paralelo. Es la que se muestra en la ilustración. El PIO, como las restantes pastillas del sistema, se conecta a los tres buses y se dispone al menos de dos *ports* (accesos) de 16 bits para comunicación con el mundo exterior. Para más detalles sobre el funcionamiento de un PIO real véase el libro "Del chip al sistema", del mismo autor, o bien para detalles específicos del sistema 6502, ver el capítulo 7 (Dispositivos de entrada/salida).

Todas estas pastillas se conectan a los tres buses, incluyendo el bus de control. No obstante, para esclarecer la ilustración, las conexiones entre el bus de control y estas diversas pastillas no se muestran en el diagrama.

Los módulos funcionales que se han descrito no residen generalmente en una sola pastilla LSI. De hecho, utilizaremos *combinaciones de pastillas* que incluyen un PIO y una cantidad limitada de ROM o RAM. Para más detalles ver el capítulo 7.

Se requieren todavía más componentes para construir un sistema real. En general, los buses suelen necesitar ser *amplificados*. Se puede utilizar también una *lógica de decodificación* para las pastillas de memoria RAM y finalmente, ciertas señales pueden necesitar ser amplificadas por *excitado-*

res o controladores ("drivers"). Estos circuitos auxiliares no se describirán aquí ya que no intervienen en la programación.

ORGANIZACIÓN INTERNA DEL 6502

Un diagrama simplificado de la organización interna del 6502 aparece en la figura 2-2.

La unidad aritmética y lógica (ALU) aparece a la derecha de la ilustración. Se la puede reconocer fácilmente por su forma de "V" característica. La función de la ALU es ejecutar operaciones aritméticas y lógicas con los datos que se envían a través de sus dos ports de entrada. Los dos ports de entrada de la ALU son respectivamente la "entrada izquierda" y la "entrada derecha". Corresponden a los dos extremos superiores de la forma "V". Después de realizar una operación aritmética, como una suma o una resta, la ALU saca su contenido por la parte inferior de dicha "V".

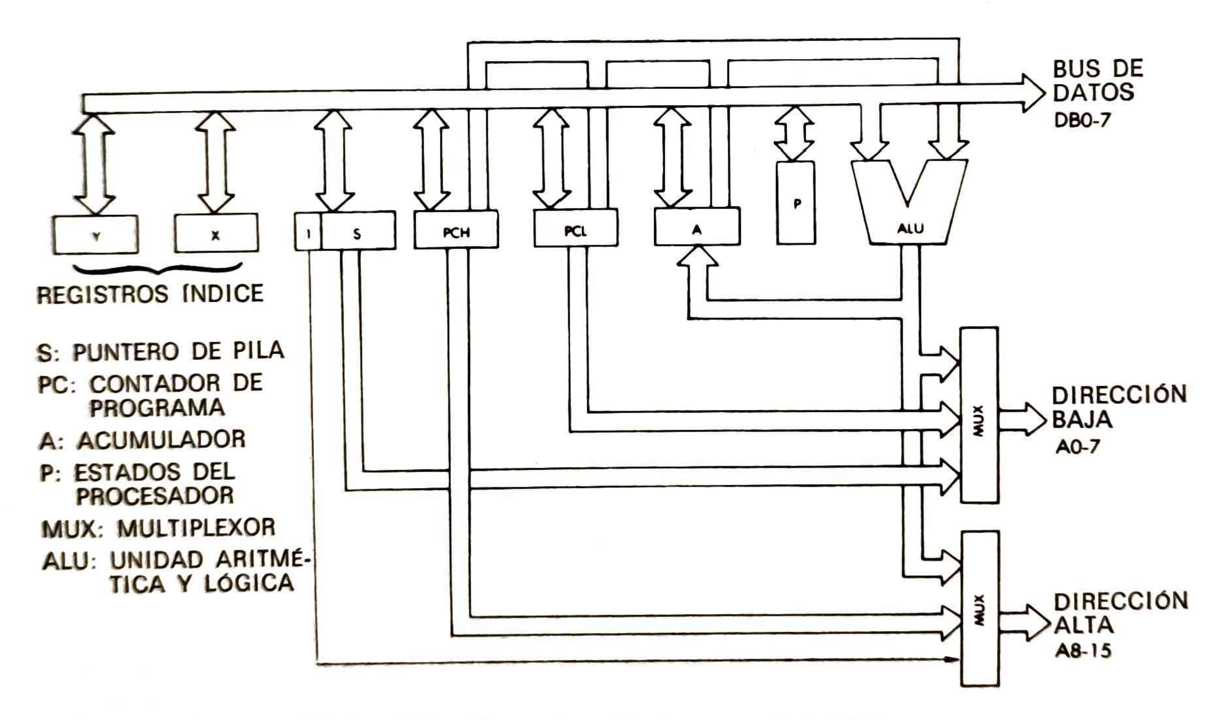


Figura 2-2 Organización interna del 6502.

La ALU está equipada con un registro especial, el *acumulador* (A). El acumulador está encima de la entrada izquierda. La ALU se relaciona automáticamente con este acumulador como una de sus entradas. (Sin embargo, existe también un modo de cortocircuitarlo.) Este es un diseño clásico basado en el acumulador. En operaciones aritméticas y lógicas, uno de los operan-

dos será el acumulador, y el otro será normalmente una posición de memoria. El resultado se depositará en el acumulador. La referencia al acumulador tanto como fuente como en funciones de destino de datos es la razón de su nombre: acumula los resultados. La ventaja de esta configuración, basada en el acumulador, es la posibilidad de utilizar instrucciones muy cortas de un solo byte (8 bits) para especificar el "código de operación", o sea, la naturaleza de la operación realizada. Si el operando se tiene que buscar y cargar en uno de los restantes registros (distintos del acumulador), será necesario utilizar un número de bits adicionales para designar a este registro dentro de la instrucción. Por consiguiente, la arquitectura del acumulador mejora la velocidad de ejecución. La desventaja es que el acumulador se debe siempre cargar con los datos deseados antes de su uso. Ello puede ocasionar ineficacias.

Veamos de nuevo la ilustración. Al lado de la ALU, a su izquierda, aparece un registro particular de 8 bits, el registro de los *indicadores de estado* del procesador (P). Este registro contiene 8 bits de estado. Cada uno de estos bits, realizado físicamente mediante un flip-flop en el interior del registro, se utiliza para indicar una condición particular. La función de los diferentes bits de estado se explicará, de modo progresivo, durante los ejemplos de programación presentados en el siguiente capítulo y serán descritos completamente en el capítulo 4, en el que se presenta el juego de instrucciones completo. Por ejemplo, tres de tales bits indicadores de estado son N, Z y C.

N representa "negativo". Es el bit 7 (o sea, el más a la izquierda) del registro P. Cuando este bit es uno, se indica que el resultado de la operación en la ALU es negativo.

El bit Z representa cero. Siempre que este bit (bit de posición 1) es un uno, significa que se obtiene un resultado cero.

El bit C, en la posición más a la derecha (posición 0) es un bit de *acarreo*. Siempre que dos números de 8 bits se suman y el resultado no se puede contener en 8 bits, el bit C es el noveno bit del resultado. El acarreo se utiliza mucho durante las operaciones aritméticas.

Estos bits de estado se posicionan automáticamente por las diferentes instrucciones. Una lista completa de las instrucciones, y el modo en que ellas afectan a los bits de estado del sistema, aparecen en el apéndice B, así como en el capítulo 4. Estos bits serán utilizados por el programador para comprobar diferentes condiciones particulares o excepcionales, o bien para comprobar rápidamente si un resultado es erróneo. Por ejemplo, la comprobación del bit Z se puede realizar con instrucciones especiales y permitirán inmediatamente saber si el resultado de una operación anterior fue 0, o no. Todas las decisiones de un programa en lenguaje ensamblador (como en todos los programas que se desarrollarán en este libro) se basarán en la

comprobación de bits. Estos bits serán bits objeto de lectura desde el mundo exterior, o bien los bits de estado de la ALU. Es, pues, muy importante comprender la función y uso de todos los bits de estado del sistema. La ALU está aquí provista de un registro que contiene estos bits. Las restantes pastillas de entrada/salida del sistema estarán también provistas de bits de estado. Éstos se estudiarán en el capítulo 7.

Desplacémonos ahora hacia la izquierda de la ALU en la figura 2-2. Los rectángulos horizontales representan los registros internos del 6502.

PC es el *contador de programa*. Es un registro de 16 bits y está realizado físicamente como dos registros de 8 bits: PCL y PCH. PCL (contador de programa inferior) representa la mitad baja del contador de programa; esto es, bits 0 a 7. PCH (contador de programa superior) representa la mitad alta del contador; esto es, los bits 8 a 15. El contador de programa es un registro de 16 bits que contiene la dirección de la siguiente instrucción a ejecutar. Cada ordenador está dotado con contador de programa, de modo que sabe qué instrucción se va a ejecutar a continuación. Revisemos brevemente el mecanismo de acceso a memoria para ilustrar la misión del contador de programa.

CICLO DE EJECUCIÓN DE UNA INSTRUCCIÓN

Veamos ahora la figura 2-3. El microprocesador aparece a la izquierda y la memoria a la derecha. La pastilla de memoria puede ser una ROM o una RAM, o cualquier otra pastilla que haya de contener memoria. La memoria se utiliza para contener instrucciones y datos. Aquí, buscaremos y cargaremos una instrucción desde la memoria para ilustrar la misión del contador de programa. Supongamos que el contador de programa tiene un contenido válido. Contiene ahora una dirección de 16 bits, que es la direc-

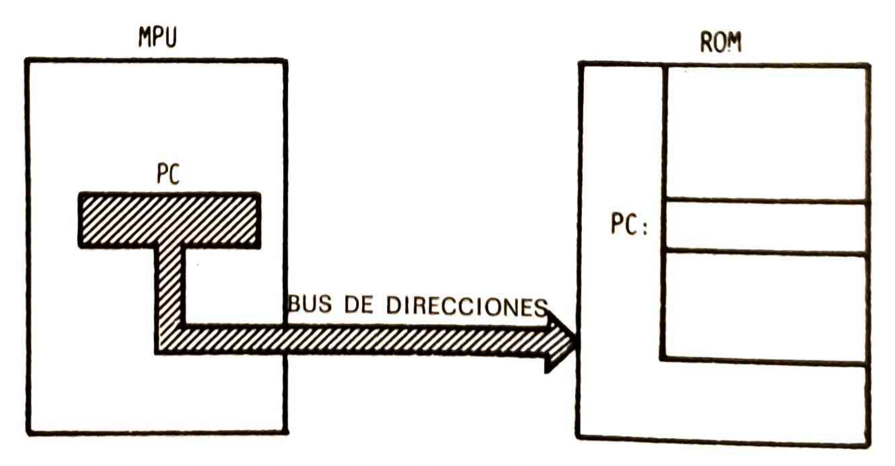


Figura 2-3 Búsqueda y carga de una instrucción de la memoria.

ción de la siguiente instrucción a buscar y cargar en la memoria. Todo procesador procede en tres ciclos:

1 — Buscar y cargar la instrucción siguiente
2 — Decodificar la instrucción
3 — Ejecutar la instrucción

Búsqueda y carga

Continuemos ahora la secuencia. En el primer ciclo, el contenido del contador de programa se deposita en el bus de direcciones y se envía a la memoria (en el bus de direcciones). Simultáneamente, una señal de lectura puede salir del bus de control del sistema, si se necesita. La memoria recibirá la dirección. Esta dirección se utiliza para especificar una posición en la memoria. Al recibir la señal de lectura, la memoria decodificará la dirección que ha recibido, en sus decodificadores internos, y seleccionará la posición indicada por la dirección. Transcurridos unos pocos centenares de nanosegundos, la memoria deposita en el bus de datos los 8 bits de datos correspondientes a la dirección indicada por su bus de datos. Esta palabra de 8 bits es la instrucción que queremos buscar y cargar. En nuestra ilustración (fig. 2-4), esta instrucción se depositará en el bus de datos (en la parte superior).

Resumamos brevemente la secuencia. El contenido del contador de programa se extrae del bus de direcciones. Se genera una señal de lectura. La memoria completa un ciclo. Aproximadamente unos 300 nanosegundos más tarde, la instrucción en la dirección indicada se deposita en el bus de datos. El microprocesador lee entonces el bus de datos y deposita su contenido en un registro interno especial, el registro IR, que es el *registro de instrucciones*. Tiene 8 bits de longitud y se utiliza para contener la instrucción que se acaba de buscar y cargar de la memoria. El ciclo de búsqueda y carga se termina ahora. Los 8 bits de la instrucción están ahora físicamente en el registro interno especial del 6502, el registro IR, el cual aparece en la parte izquierda de la figura 2-4.

Decodificación y ejecución

Una vez que la instrucción esté contenida en IR, la unidad de control del microprocesador decodificará el contenido y podrá generar la secuencia correcta de señales, interna y externa, para la ejecución de la instrucción específica. Hay, pues, un retardo de decodificación corto seguido por una fase de ejecución, cuya longitud depende de la naturaleza de la instrucción especificada. Algunas instrucciones se ejecutarán completamente dentro de la MPU. Otras instrucciones buscarán y cargarán o depositarán datos desde,

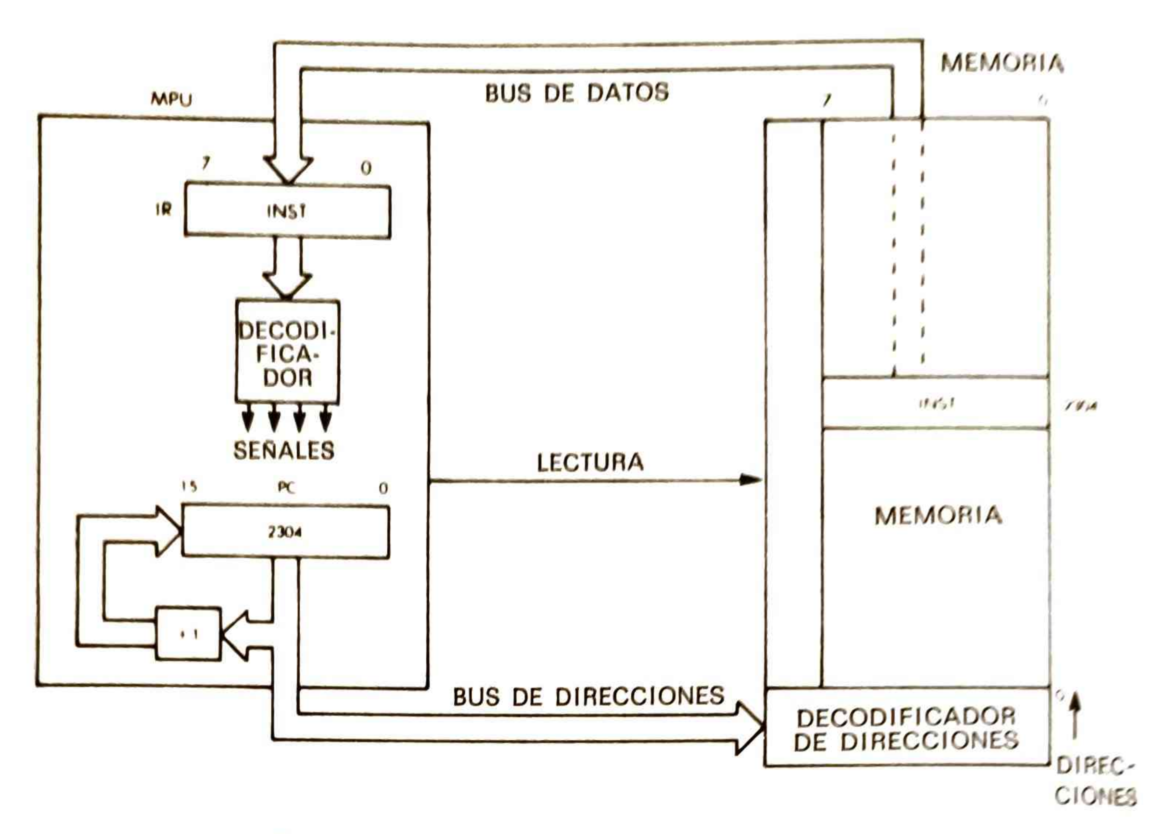


Figura 2-4 Control secuencial automático.

o en, la memoria. Esto es así porque las diferentes instrucciones del 6502 tienen duraciones de ejecución distintas. Esta duración se expresa como un número de ciclos de reloj. El apéndice D da el número de ciclos requeridos por cada instrucción. Un 6502 normal utiliza un reloj de 1 megahercio. La longitud de cada ciclo es, por tanto, 1 microsegundo. Ya que se pueden utilizar diferentes frecuencias de reloj con componentes diferentes, la velocidad de ejecución suele expresarse en número de ciclos, en vez de en número de nanosegundos.

En el caso del 6502, su *reloj* es interno y se representa por el oscilador interno (fig. 2-1).

Búsqueda y carga de la siguiente instrucción

Ahora hemos descrito cómo, utilizando el contador de programa, se puede buscar y cargar una instrución de la memoria. Durante la ejecución de un programa, las instrucciones se buscan y cargan *en secuencia* desde la memoria. Es preciso, pues, un mecanismo automático para proporcionar la búsqueda y carga de las instrucciones en secuencia. Esta tarea se realiza por un sencillo incrementador conectado al contador de programa. Esto se ilustra en la figura 2-4. Cada vez que el contenido del contador de programa (en la

parte inferior de la ilustración) se sitúa en el bus de direcciones, su contenido se incrementará y se escribirá, de nuevo, en el contador de programa. Por ejemplo, si el contador de programa contiene el valor 0, se extraerá el valor 0 en el bus de direcciones. A continuación se incrementará el contenido del contador de programa y el valor 1 se escribirá de nuevo en el contador de programa. De este modo, la próxima vez que se utilice el contador de programa, será la instrucción en la dirección 1 la que se obtendrá. Acabamos de realizar un mecanismo automático para secuenciar las instrucciones.

Se debe insistir en que las descripciones anteriores son simplificadas. En realidad, algunas instrucciones pueden ser de 2 o incluso 3 bytes de longitud, de modo que los bytes sucesivos se puedan buscar y cargar de este modo desde la memoria. Sin embargo, el mecanismo es idéntico. El contador de programa se utiliza para buscar y cargar bytes sucesivos de una instrucción, así como para buscar y cargar instrucciones sucesivas por sí mismas. El contador de programa, junto con su incrementador, proporciona un mecanismo automático para apuntar a posiciones de memoria sucesivas.

Otros registros del 6502

Una última parte de la figura 2-2 no ha sido explicada todavía. Es el conjunto de tres registros rotulados X, Y y S. Los registros X e Y se denominan registros índice. Tienen 8 bits de longitud. Se pueden utilizar para contener los datos con los cuales funcionará el programa. Sin embargo, suelen emplearse como registros índice.

El papel de los registros índice se describirá en el capítulo 5 sobre técnicas de direccionamiento. Digamos brevemente, que el contenido de estos dos registros índice se pueden sumar de varios modos a cualquier dirección especificada dentro del sistema para proporcionar un desplazamiento automático. Es ésta una posibilidad importante para recuperar datos eficazmente cuando están almacenados en tablas. Estos dos registros no son completamente iguales y sus funciones se diferenciarán en el capítulo de técnicas de direccionamiento.

El registro de pila (stack) S se utiliza para contener un puntero hacia la parte superior de la zona de la pila en la memoria.

Introduzcamos ahora el concepto formal de pila.

LA PILA

Una pila se denomina formalmente estructura LIFO (last-in, first-out = último en entrar, primero en salir). Una pila es un conjunto de registros,

o posiciones de memoria, asignados a esta estructura de datos. La característica esencial de esta estructura es que se trata de una estructura *cronológica*. El primer elemento introducido en la pila está siempre en la parte inferior de la misma. El elemento depositado más recientemente en la pila está en la cima de la pila. Se puede trazar una analogía con una pila de platos en un mostrador de un restaurante. Hay un agujero en el mostrador con un muelle en el fondo. Los platos son apilados encima del agujero. Con esta disposición, se garantiza que el plato que se ha puesto primero en la pila (el que lleva más tiempo) está siempre en el fondo. El que se ha situado en la pila más recientemente es el que está encima de él. Este ejemplo ilustra otra característica de la pila. En utilización normal, una pila es accesible solamente por medio de dos instrucciones: "meter" y "sacar" (o "extraer"). La operación *meter* (push) permite depositar un elemento en la cima de la pila. La operación *extraer* (pull) consiste en quitar un elemento de la pila. En la práctica, en el caso de un microprocesador, el *acumulador* es el que se depositará en la parte superior de la pila. La operación *sacar* (pop) obtendrá una transferencia del elemento de la cima de la pila al acumulador. Otras instrucciones especializadas pueden existir para transferir la cima de la pila entre otros registros especializados, tales como el registro de estados.

La presencia de una pila se requiere para poner en práctica tres medios de programación en el sistema de ordenador: subrutinas, interrupciones y almacenamiento de datos temporales. La función de la pila durante las subrutinas se explicará en el capítulo 3 (Técnicas de programación básicas). La función de la pila durante las interrupciones se explicará en el capítulo 6 (Técnicas de entradas/salidas). Finalmente, la función de la pila para salvaguardar datos a gran velocidad se explicará durante los programas de aplicación específicos.

Ahora supondremos, simplemente, que la pila es un elemento requerido en todo sistema de ordenador.

Una pila se puede poner en práctica de dos maneras:

1. Un número fijo de registros se pueden proporcionar dentro del propio microprocesador. Esto es una "pila de hardware". Tiene la ventaja de la alta velocidad. Sin embargo, tiene el inconveniente de un número limitado de registros.

2. En la mayoría de los microprocesadores de uso general se utiliza otro método, el de la pila de software, para no limitar la pila a un número muy pequeño de registros. Esta es la configuración elegida en el 6502. En la configuración de software, un registro especial dentro del microprocesador, en nuestro caso el registro S, almacena el puntero de pila, es decir, la dirección del elemento de la cima de la pila (o más exactamente, esta dirección más uno). La pila se realiza, pues, como una zona de memoria. El puntero de pila exigirá, por tanto, 16 bits para apuntar a cualquier parte de la memoria.

Sin embargo, en el caso del 6502, el puntero de pila se limita a 8 bits. Se incluye un 9.° bit, en la posición más a la izquierda, siempre puesto a 1. En otras palabras, la zona asignada a la pila en el caso del 6502 va desde la dirección 256 a la 511. En binario, es desde "100000000" a "111111111". La pila empieza siempre en la dirección 111111111 y puede tener hasta 255 palabras. Ello se puede considerar como una limitación del 6502 y se comentará más adelante en este libro. En el 6502 la pila comienza siempre en la dirección más alta y crece "hacia atrás"; el puntero de pila se decrementa mediante un PUSH.

Para utilizar la pila, el programador sólo tendrá que inicializar el registro S. El resto es automático.

Se dice que la pila reside en la *página 1* de la memoria. Introduzcamos, ahora, el concepto de *paginación*.

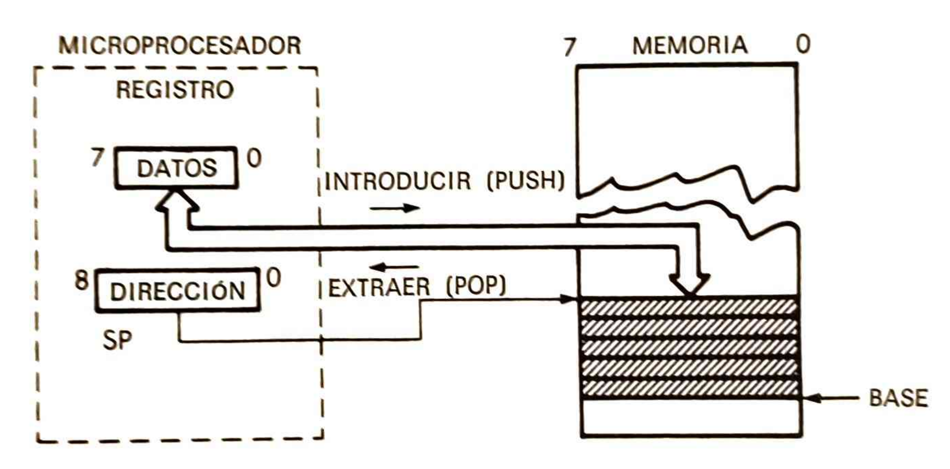


Figura 2-5   Las dos instrucciones de manipulación de la pila.

EL CONCEPTO DE PAGINACIÓN

El microprocesador 6502 posee un bus de direcciones de 16 bits. Los 16 bits se pueden utilizar para crear hasta $2^{16} = 64K$ combinaciones (1K igual a 1.024). Habida cuenta de las características de direccionamiento del 6502 que se presentarán en el capítulo 5, es conveniente la partición de la memoria en *páginas* lógicas. Una página no es más que un bloque de 256 palabras. Así, las posiciones de memoria 0 a 255 son la página 0 de la memoria. Se utilizará para el modo de direccionamiento por "página cero". La página 1 de la memoria incluye las posiciones de memoria 256 a 511. Hemos establecido que la página 1 suele reservarse para la zona de la pila. Todas las demás páginas del sistema se consideran para el diseño y se pueden

utilizar como se quiera. En el caso del 6502, es importante conservar la organización de páginas de la memoria. Cuando haya de franquearse una frontera de página, se introducirá, a menudo, un retardo suplementario de un ciclo en la ejecución de una instrucción.

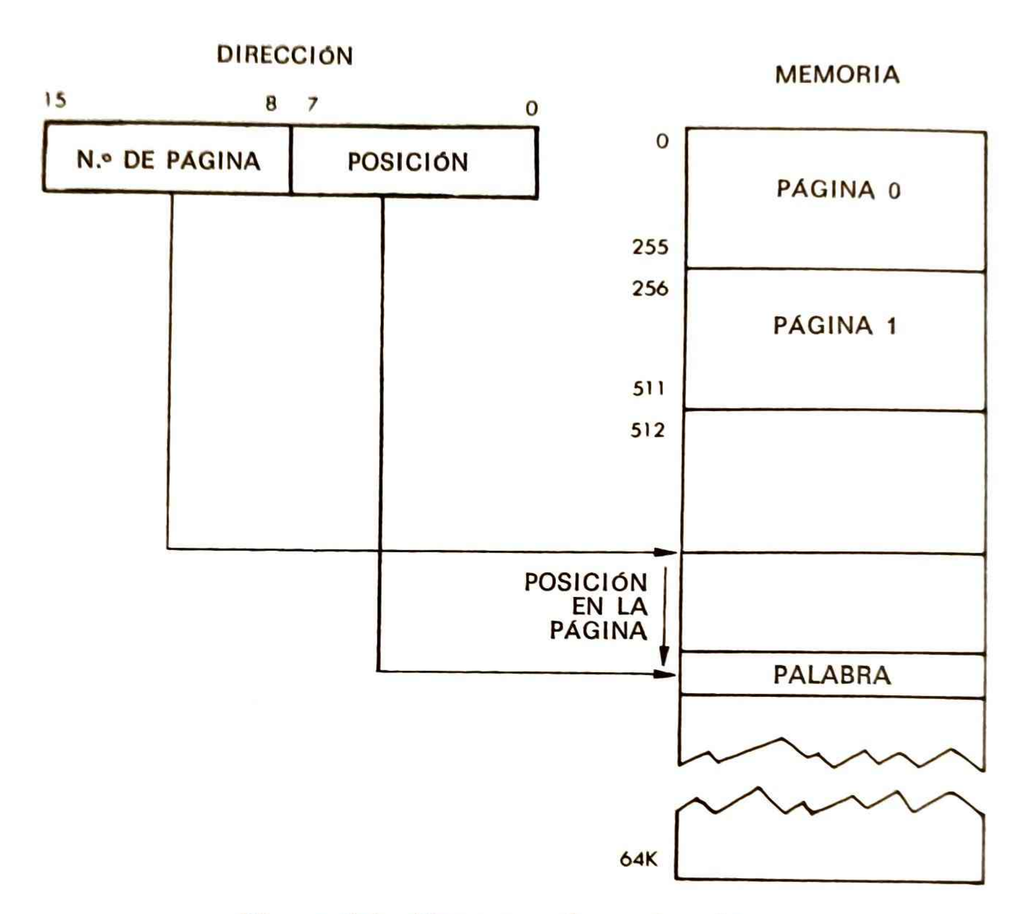


Figura 2-6 Concepto de paginación.

LA PASTILLA DEL 6502

Para completar nuestra descripción del diagrama, el bus de datos en la parte superior de la figura 2-2 representa el bus de datos externo. Se utilizará para comunicar con los dispositivos externos y la memoria, en particular. A0-7 y A8-15 representan respectivamente la parte de orden-bajo y la parte de orden-alto del bus de direcciones creado por el 6502.

Para ser completos, presentamos aquí el diagrama de conexiones exacto del microprocesador 6502. No necesita leerlo para comprender el resto de este libro. Sin embargo, si trata de conectar dispositivos a un sistema, esta descripción será necesaria.

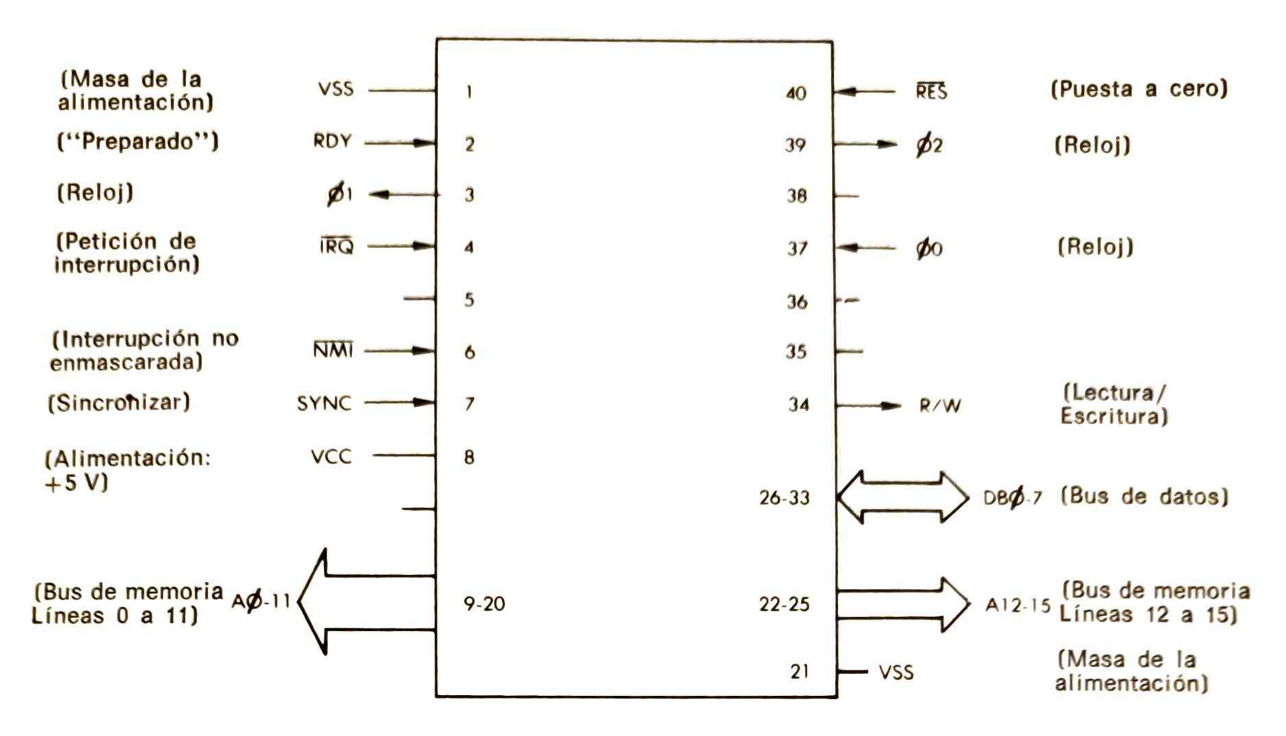


Figura 2-7 Diagrama de conexiones del 6502.

El diagrama de conexiones del 6502 aparece en la figura 2-7. El bus de datos se representa por DB0-7 y es reconocible fácilmente a la derecha de la ilustración. El bus de direcciones se representa por A0-11 y A12-15 y comprende las patillas 9 a 20 en la izquierda de la pastilla y las patillas 22 a 25 a su derecha.

El resto de las señales son de alimentación y de control.

Las señales de control

— R/W: es la línea de LECTURA/ESCRITURA que controla la dirección de las transferencias de datos en el bus de datos.
— $\overline{\text{IRQ}}$ y $\overline{\text{NMI}}$ son *"petición de interrupción"* e *"interrupción no enmascarada"*. Son dos líneas de interrupción y se utilizarán en el capítulo 7.
— SYNC es una señal que indica una búsqueda y carga del código de operación para el mundo exterior.
— RDY suele emplearse para sincronizar una memoria lenta: parará el procesador.
— SO pone a uno el indicador de desbordamiento. No suele utilizarse.
— $\emptyset_0$, $\emptyset_1$, y $\emptyset_2$ son señales de reloj.
— $\overline{\text{RES}}$ es la puesta a cero (RESET) que se utiliza para inicializar.
— V_{SS} y V_{CC} son las patillas de alimentación (5 V).

RESUMEN DE HARDWARE

Con lo anterior se termina nuestra descripción de hardware de la organización interna del 6502. La estructura exacta de los buses internos del 6502 no es importante aquí. Sin embargo, la función exacta de cada uno de los registros sí es importante y se deberá comprender completamente antes de continuar. Si está familiarizado con los conceptos que se han presentado, continúe con los capítulos siguientes. Si no se siente seguro respecto a alguno de ellos, se le recomienda que lea de nuevo las secciones importantes de este capítulo, ya que serán necesarias en los próximos. Se le recomienda que vuelva a observar la figura 2-2 y se cerciore de que comprendió la función de cada registro en la ilustración.

3 Técnicas de programación básicas

INTRODUCCIÓN

El objetivo de este capítulo es presentar todas las técnicas básicas necesarias para escribir un programa empleando el 6502. Este capítulo introducirá conceptos adicionales tales como la gestión de registros, bucles y subrutinas (subprogramas). Se pondrá atención en las técnicas de programación utilizando solamente los recursos *internos* del 6502, o sea, los registros. Se desarrollarán programas prácticos tales como programas aritméticos. Estos programas servirán para ilustrar los diversos conceptos presentados hasta aquí y utilizaremos instrucciones verdaderas. Así, se verá cómo se pueden utilizar instrucciones para tratar la información entre la memoria y la MPU, así como para tratar la información dentro de la MPU propiamente dicha. En el capítulo siguiente se describirán detalladamente las instrucciones disponibles en el 6502. En el capítulo 6 se presentarán las técnicas disponibles para tratar la información *fuera* del 6502, es decir, las técnicas de entrada/salida.

En este capítulo aprenderemos básicamente por la práctica. Examinando programas de complejidad creciente, se aprenderá la función de las diferentes instrucciones y de los registros y se aplicarán los conceptos desarrollados hasta este momento. No obstante, un concepto importante no estará presente aquí: el de las técnicas de direccionamiento. A causa de su aparente complejidad, se presentará por separado en el capítulo 5.

Comencemos a escribir inmediatamente algunos programas para el 6502. Los iniciaremos con programas aritméticos.

PROGRAMAS ARITMÉTICOS

Los programas aritméticos son esencialmente suma, resta, multiplicación y división. Los programas que se presentarán aquí se referirán a enteros. Estos enteros pueden ser enteros binarios positivos o se pueden expresar en notación de complemento a dos, en cuyo caso el bit más a la izquierda es el bit de signo (ver el capítulo 1 para recordar la notación del complemento a dos).

Suma de 8 bits

Sumaremos dos operandos de 8 bits llamados OP1 y OP2, almacenados respectivamente en las direcciones de memoria ADR1 y ADR2. La suma se denominará RES y se almacenará en la dirección de memoria ADR3. Esto se ilustra en la figura 3-1. El programa que efectuará esta suma es el siguiente:

```
LDA    ADR1    CARGAR OP1 EN A
ADC    ADR2    SUMAR OP2 A OP1
STA    ADR3    GUARDAR RES EN ADR3
```

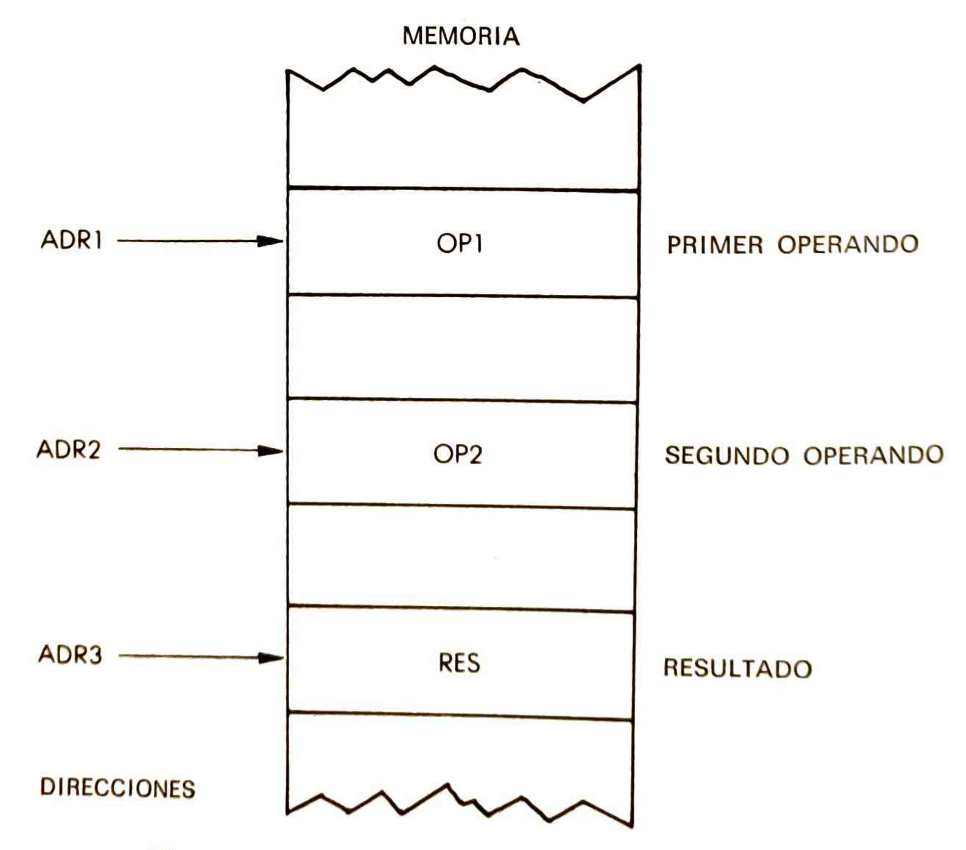


Figura 3-1 Suma de 8 bits. RES = OP1 + OP2.

Este es un programa de tres instrucciones. Cada línea es una instrucción, en forma simbólica. Cada una de estas instrucciones será traducida por el programa ensamblador en 1, 2 o 3 bytes binarios. No nos preocuparemos aquí por la traducción y solamente consideramos la representación simbólica. La primera línea es una instrucción LDA que significa "cargar el acumulador A con el contenido de la dirección que sigue".

La dirección especificada en la primera línea es ADR1. Ésta es una representación simbólica de una dirección real de 16 bits. En otra parte del programa, se definirá el símbolo ADR 1. Podrá ser, por ejemplo, la dirección 100.

La instrucción LDA indica "cargar el acumulador A" (en el interior del 6502) desde la posición de memoria 100. Resultará de ello una operación de lectura de la dirección 100, cuyo contenido se transmitirá a lo largo del bus de datos y se deposita en el acumulador. Hay que tener presente que las operaciones aritméticas y lógicas consideran al acumulador como uno de los operandos fuente. (Consultar el capítulo anterior para más detalles.) Ya que deseamos sumar los dos valores OP1 y OP2 juntos, cargamos en primer lugar OP1 en el acumulador. A continuación podremos sumar los contenidos del acumulador (OP1) a OP2.

El campo más a la derecha de esta instrucción se llama *campo de comentarios.* Se ignora por el procesador, pero sirve para facilitar la lectura del programa. Para comprender qué hace el programa, es de gran importancia utilizar comentarios adecuados. A esto se le denomina *documentar* un programa. Aquí el comentario es evidente: el valor de OP1, que se encuentra en la dirección ADR1, es cargado en el acumulador A.

El resultado de esta primera instrucción se ilustra en la figura 3-2.

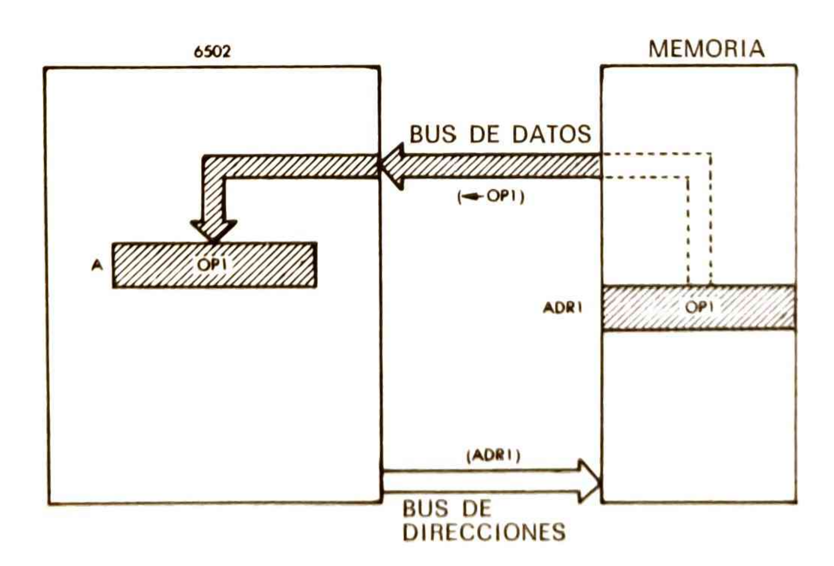


Figura 3-2 LDA ADR1: OP1 se carga desde la memoria.

La segunda instrucción de nuestro programa es:

ADC ADR2

Ella indica "sumar el contenido de la posición de memoria ADR2 al acumulador". Con referencia a la figura 3-1, el contenido de la posición de memoria ADR2 es OP2, su segundo operando. El contenido actual del acumulador es ahora OP1, nuestro primer operando. Como resultado de la ejecución de la segunda instrucción, OP2 se buscará y cargará en la memoria y se sumará a OP1. La suma se depositará en el acumulador. El lector recordará que los resultados de una operación aritmética, en el caso del 6502, se depositan, de nuevo, en el acumulador. En otros microprocesadores, puede ser posible depositar este resultado en otros registros o, de nuevo, en la memoria.

La suma de OP1 y OP2 está ahora en el acumulador. Hemos transferido el contenido del acumulador a la posición de memoria ADR3 para almacenar el resultado en la posición especificada. De nuevo, el campo más a la derecha de la segunda instrucción es simplemente un campo de comentarios que describe la función de la instrucción (sumar OP2 a A). El efecto de la segunda instrucción se observa en la figura 3-3.

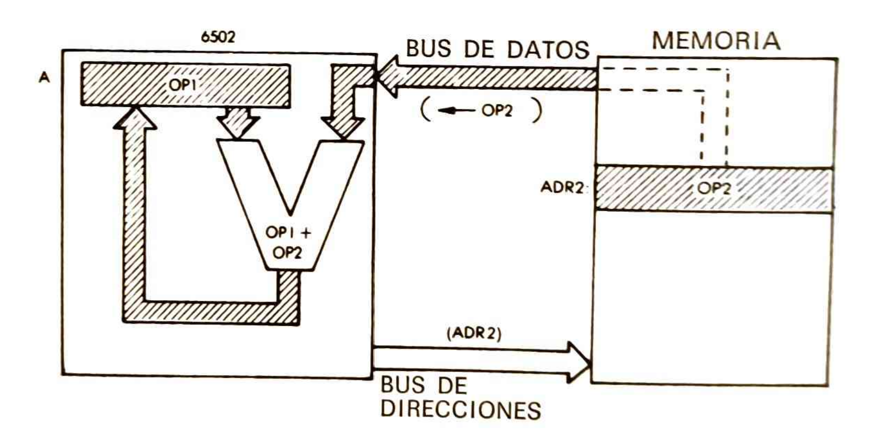


Figura 3-3 ADC ADR2.

Se puede verificar en la figura 3-3 que, inicialmente, el acumulador contiene OP1. Después de la suma, un nuevo resultado se ha escrito en el acumulador. Es OP1 + OP2. El contenido de cualquier registro en el sistema, así como cualquier posición de memoria, permanece inalterado cuando se ejecuta una operación de lectura. En otras palabras, *la lectura de un registro o una posición de memoria no altera su contenido*. Es sólo, y exclusivamente, una operación de escritura lo que modificará el contenido de un registro. En este ejemplo, los contenidos de las posiciones de memoria ADR1 y ADR2

permanecen inalterados. Sin embargo, después de la segunda instrucción de este programa, el contenido del acumulador se ha modificado porque la salida de la ALU se ha escrito en el acumulador. Entonces, se perderá su contenido anterior.

Conservemos ahora este resultado en la dirección ADR3 y habremos completado nuestra suma elemental.

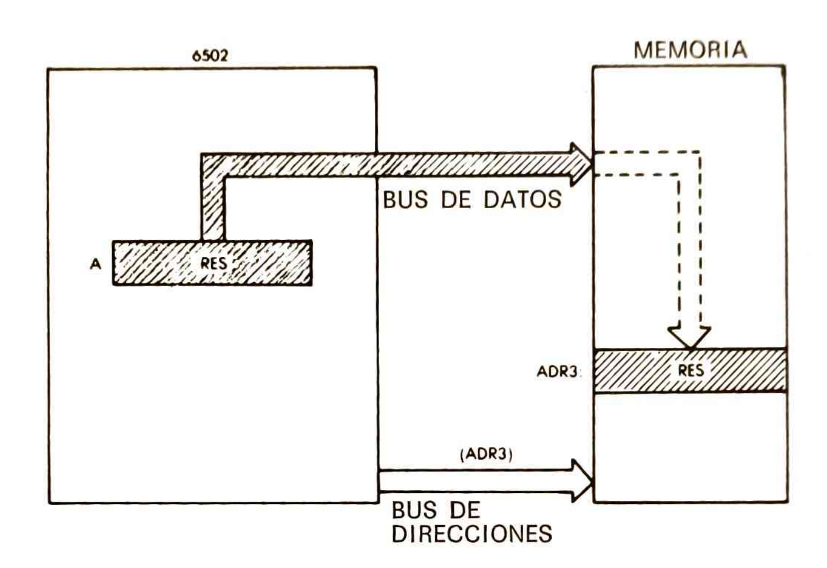


Figura 3-4 STA ADR3 (Guardar acumulador en memoria).

La tercera instrucción es STA ADR3, lo que significa "almacenar el contenido del acumulador A en la dirección ADR3". Ello se explica por sí mismo y se ilustra en la figura 3-4.

Peculiaridades del 6502

El anterior programa de tres instrucciones constituirá efectivamente el programa completo de la mayoría de los microprocesadores. Sin embargo, el 6502 tiene dos peculiaridades que suelen requerir dos instrucciones adicionales.

En primer lugar, la instrucción ADC significa realmente "sumar *con acarreo*", más que "sumar". La diferencia es que una simple instrucción de suma hace que se efectúe una suma de dos números. En cambio, una suma con acarreo suma dos números más el valor del bit de acarreo. Ya que sumamos números de 8 bits, no se utilizará ningún acarreo y, en el momento en que comenzamos la suma, no conocemos necesariamente el estado del bit de acarreo (puede haber sido "puesto a uno" por la instrucción anterior),

por lo que debemos borrarlo, o, lo que es lo mismo, ponerlo a cero. Esto se realizará por la instrucción CLC: "borrar acarreo".

Lamentablemente, el 6502 no tiene los dos tipos de operaciones de suma. No tiene más que una operación ADC. Resulta de ello que, para sumas sencillas de 8 bits, es necesario siempre tomar la precaución de borrar el bit de acarreo. Aunque esto no constituye una desventaja, no debe ser olvidado.

La segunda peculiaridad del 6502 reside en el hecho de que dispone de instrucciones decimales potentes, que se utilizarán en la próxima sección de aritmética BCD. El 6502 funciona siempre en uno de dos modos: binario o decimal. El estado en que se encuentra está determinado por un bit de estado, el bit «D» (del registro P). Ya que estamos trabajando en modo binario en este ejemplo, es preciso cerciorarse de que el bit D está correctamente posicionado. Esto se hará por una instrucción CLD, la cual borrará el bit D. Naturalmente, si toda la aritmética del sistema se hace en binario, el bit D será puesto a cero, de una vez por todas, al principio del programa y no será necesario hacerlo cada vez. En consecuencia, esta instrucción puede ser omitida en la mayoría de los programas. Sin embargo, el lector que practique estos ejercicios en un ordenador puede alternar los ejercicios binarios y en BCD, y esta instrucción adicional que se ha incluido aquí debe aparecer al menos una vez, antes de que se realice cualquier operación binaria.

En resumen, nuestro programa de suma de 8 bits completo y seguro es ahora:

```
CLC                BORRAR BIT DE ACARREO
CLD                BORRAR BIT DE MODO DECIMAL
LDA     ADR1       CARGAR OP1 EN A
ADC     ADR2       SUMAR OP2 A OP1
STA     ADR3       ALMACENAR RES EN ADR3
```

Se pueden utilizar direcciones físicas reales en vez de ADR1, ADR2 y ADR3. Si se desean guardar direcciones simbólicas, será necesario utilizar las denominadas "pseudoinstrucciones" que especifican el valor de estas direcciones simbólicas de modo que el programa ensamblador puede, durante su traducción, sustituir las direcciones físicas reales. Tales pseudoinstrucciones serán, por ejemplo:

```
ADR1 = $ 100
ADR2 = $ 120
ADR3 = $ 200
```

Ejercicio 3.1: *Ahora cierre el libro. Consulte solamente la lista de instrucciones al final del libro. Escriba un programa que sume dos números almacenados en las posiciones de memoria LOC1 y LOC2. Deposite el resultado*

en la posición de memoria LOC3. A continuación, compare su programa con el programa anterior.

Suma de 16 bits

La suma de 8 bits no permite más que la suma de números de 8 bits; esto es, números entre 0 y 255, si se utiliza binario absoluto. Para la mayoría de las aplicaciones prácticas es necesario utilizar *multiprecisión* y sumar números que tengan 16 bits o más. Presentaremos ahora ejemplos de aritmética de números de 16 bits. Ellos pueden ampliarse fácilmente a 24, 32 o más bits (se utilizan siempre múltiplos de 8 bits). Supondremos que el primer operando se almacena en las posiciones de memoria ADR1 y ADR1 — 1. Como OP1 es ahora un número de 16 bits, requerirá dos posiciones de memoria de 8 bits. De modo similar, OP2 se almacenará en ADR2 y ADR2 — 1. El resultado se depositará en las direcciones de memoria ADR3 y ADR3 — 1. Ello se ilustra en la figura 3-5.

La lógica de este programa es semejante a la del anterior. En primer lugar, se sumará la mitad inferior de los dos operandos, ya que el microprocesador solamente puede sumar 8 bits a la vez. Cualquier acarreo generado por la suma de estos bytes de orden bajo se almacenará automática-

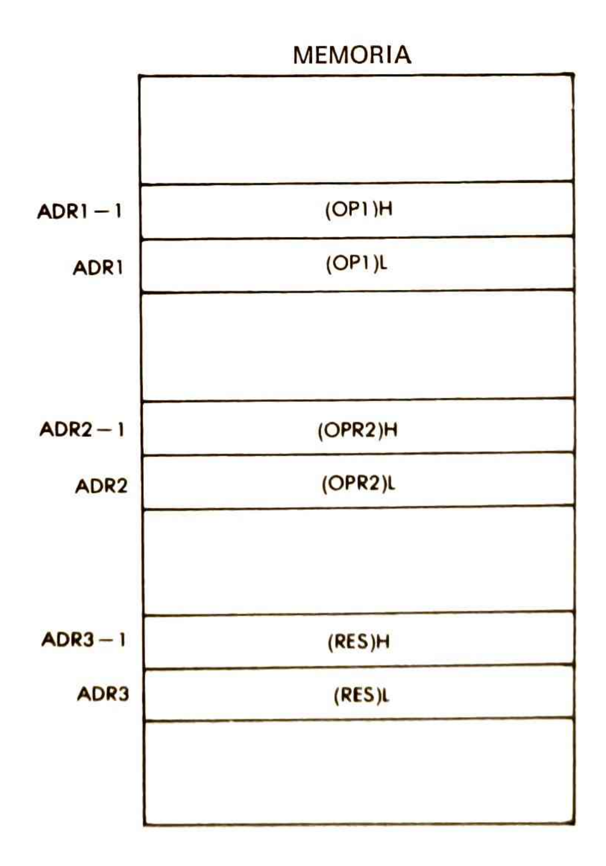


Figura 3-5 Suma de 16 bits: los operandos.

mente en el bit de acarreo interno ("C"). A continuación, la mitad de orden alto de los dos operandos se sumará junto con cualquier acarreo, y el resultado se almacenará en la memoria. El programa aparece a continuación:

```
CLC
CLD
LDA     ADR1       MITAD BAJA DE OP1
ADC     ADR2       (OP1+OP2) MITAD BAJA
STA     ADR3       ALMACENAR LA MITAD BAJA DE RES
LDA     ADR1-1     MITAD ALTA DE OP1
ADC     ADR2-1     (OP1+OP2) MITAD ALTA MAS ACARREO
STA     ADR3-1     ALMACENAR LA MITAD ALTA DE RES
```

Las dos primeras instrucciones de este programa se utilizan por seguridad: CLC, CLD. Sus funciones se explicaron en la sección anterior. Examinemos el programa. Las siguientes tres instrucciones son esencialmente idénticas a las de la suma de 8 bits del párrafo precedente. Ellas obtienen la suma de las mitades menos significativas (de menor peso, bits 0 a 7) de OP1 y OP2. La suma se llama RES y se almacena en la posición de memoria ADR3.

Automáticamente, siempre que se realiza una suma, cualquier acarreo que resulte se conserva en el bit de acarreo del registro de indicadores de estado (registro P). Si los dos números de 8 bits no generan ningún acarreo, el valor del acarreo será cero. Si los dos números generan un acarreo, entonces el bit C será igual a 1.

Las siguientes tres instrucciones del programa son también prácticamente idénticas a las del programa de suma anterior de 8 bits. Ellas suman la mitad más significativa (bit 8 a 15) de OP1 y OP2, más cualquier acarreo y almacenan los resultados en la dirección ADR3 — 1. Una vez que se haya ejecutado este programa, el resultado de 16 bits se almacena en las posiciones de memoria ADR3 y ADR3 — 1.

Se supone, en este caso, que ningún acarreo resultará de esta suma de 16 bits. Se supone que el resultado es verdaderamente un número de 16 bits. Si el programador sospecha, por cualquier motivo, que el resultado puede tener 17 bits, será preciso añadir instrucciones adicionales que comprueben el bit de acarreo después de esta suma.

La posición de los operandos en la memoria se ilustra en la figura 3-5.

Observe que en este caso hemos supuesto que la parte alta del operando se almacena «encima de» la parte inferior; esto es, en la dirección de memoria inferior. Este no es necesariamente el caso en que nos encontramos. De hecho, las direcciones se almacenan en el 6502 en modo inverso: la parte baja es la que se conserva primero en la memoria y la parte alta se conserva

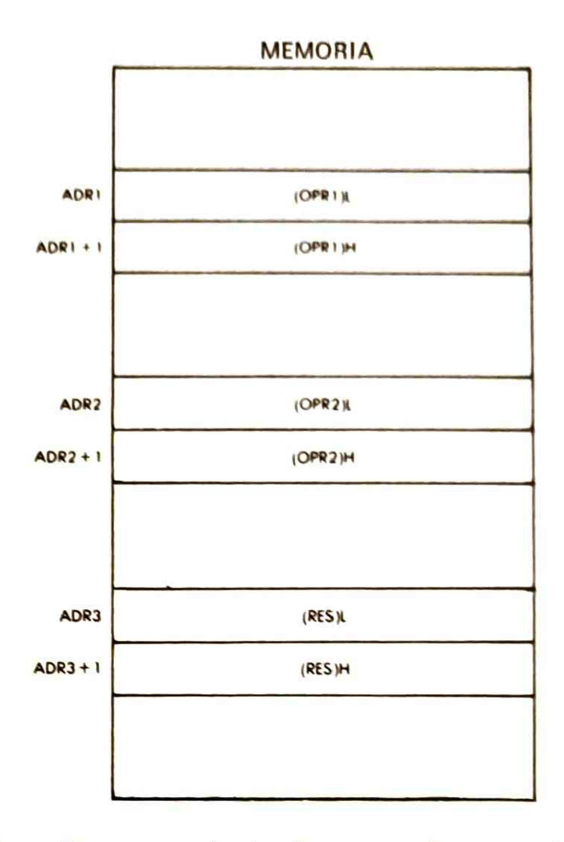


Figura 3-6a Almacenamiento de operandos en orden inverso.

en la siguiente posición de memoria. Con el fin de utilizar una notación común para direcciones y datos, se recomienda que los datos se guarden con la parte baja antes de la parte alta. Ello se ilustra en la figura 3-6*a*.

Ejercicio 3-2: *Vuelva a escribir el programa de suma de 16 bits anterior con el diagrama de memoria indicado en la figura* 3-6a.

Ejercicio 3.3: *Suponga ahora que ADR1 no apunta a la mitad inferior de OPR1 (vea la figura* 3-6a*), sino que apunta a la parte superior de OPR1. Esto se ilustra en la figura* 3-6b*. Escriba de nuevo el programa correspondiente.*

Es el programador, o sea, usted, quien debe decidir cómo almacenar números de 16 bits (parte baja o alta en primer lugar) y también si sus referencias de dirección apuntan a la mitad inferior o superior de tales números. Esta es la primera de muchas elecciones que aprenderá a realizar cuando diseñe algoritmos o estructuras de datos.

Hasta ahora hemos aprendido a ejecutar una suma binaria. Pasemos a la operación de la resta.

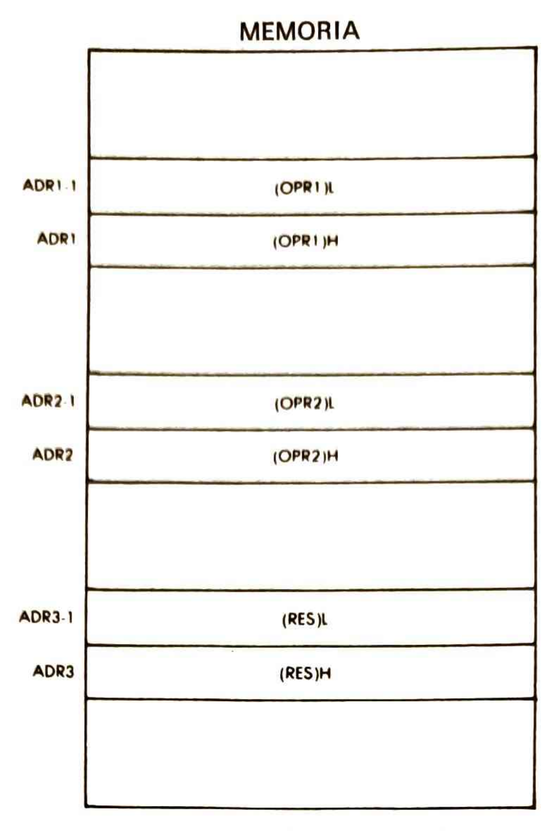


Figura 3-6b Apuntamiento hacia el byte alto.

Resta de números de 16 bits

Hacer una resta de 8 bits será demasiado sencillo. Dejémoslo como ejercicio para el lector y efectuemos directamente una resta de 16 bits. Como de costumbre, nuestros dos números OPR1 y OPR2 se almacenan en las direcciones ADR1 y ADR2. El diagrama de memoria se supondrá que sea el de la figura 3-6*a*. Para restar utilizaremos una operación de resta (SBC) en lugar de una operación de suma (ADC). El único cambio suplementario, con relación a la suma, es que utilizará una instrucción SEC al principio del programa en vez de CLC. SEC significa "poner a uno el acarreo". Esto indica una condición de "ausencia de acarreo negativo". El resto del programa es idéntico al de la suma. El programa aparece a continuación:

```
CLD
SEC                  PUESTA A UNO DEL ACARREO
LDA      ADR1        (OPR1)L (BAJO) EN A
SBC      ADR2        (OPR1)L–(OPR2)L
STA      ADR3        ALMACENAR (RESULTADO)L
LDA      ADR1+1      (OPR1)H (ALTO) EN A
SBC      ADR2+1      (OPR1)H–(OPR2)H
STA      ADR3+1      ALMACENAR (RESULTADO)H
```

Ejercicio 3.4: *Escribir el programa de resta para operandos de 8 bits.*

Es preciso tener presente que en el caso de la aritmética en complemento a dos, el valor final del indicador de acarreo no tiene significado. Si se produce una condición de desbordamiento como resultado de la resta, entonces el bit de desbordamiento (bit V) del registro de indicadores de estado se habrá puesto a uno. Entonces puede ser objeto de comprobación.

Los ejemplos que se acaban de presentar son sumas binarias sencillas. Sin embargo, puede ser necesario otro tipo de suma: la suma BCD.

Aritmética BCD

Suma BCD de 8 bits

El concepto de aritmética BCD se ha presentado en el capítulo 1. Se utiliza sobre todo en aplicaciones de gestión en donde es imperativo conservar todos los dígitos significativos en el resultado. En la notación BCD se utiliza un nibble (4 bits) para almacenar un dígito decimal (0 a 9). Resulta de ello que todos los bytes pueden almacenar dos dígitos BCD. (Esto se denomina *BCD compacto*). Sumemos ahora dos bytes que contengan dos dígitos BCD cada uno.

Para identificar los problemas examinaremos en primer lugar algunos ejemplos numéricos.

Sumemos "01" y "02":

"01" se representa por	0000 0001
"02" se representa por	0000 0010
El resultado es	0000 0011

Esta es la representación BCD de "03". (Si no está seguro del equivalente BCD, consulte la tabla de conversión en el apéndice H.) Todo ha funcionado muy sencillamente en este caso. Probemos ahora, otro ejemplo:

"08" se representa por	0000 1000
"03" se representa por	0000 0011

Ejercicio 3.5: *Calcular la suma de los dos números anteriores en la representación BCD. ¿Qué se obtiene? (la respuesta sigue a continuación).*

Si se obtiene 0000 1011, se ha calculado la suma *binaria* de "8" y "3". Hemos obtenido en su lugar "11" que es un *código ilegal en BCD*. Se deberá obtener la representación BCD de "11", o sea, "0001 0001".

El problema se plantea por el hecho de que la representación BCD utiliza solamente las primeras diez combinaciones de cuatro dígitos para codificar los símbolos "0" a "9". Las seis posibles combinaciones restantes de 4 dígitos no se emplean y la ilegal "1011" es una de ellas. En otras palabras, siempre que la suma de dos dígitos binarios sea mayor que "9", es preciso sumar "6" al resultado para saltar sobre los 6 códigos no utilizados. Sume la representación binaria "6" a "1011":

```
                       1011   (resultado binario ilegal)
                     + 0110   (+6)
                    -------
El resultado es:  0001 0001
```

Esto es, ciertamente, "11" en la notación BCD. Tenemos ahora el resultado correcto.

Este ejemplo ilustra una de las dificultades básicas del modo BCD. Es preciso compensar los seis códigos que faltan. En la mayoría de los microprocesadores, es preciso utilizar una instrucción especial, llamada "ajuste decimal", para ajustar el resultado de la suma binaria (sumar 6 si el resultado es mayor que 9). En el caso del 6502, la instrucción ADC lo hace automáticamente. Esta es una ventaja clara del 6502 cuando trabaja con aritmética BCD.

El problema siguiente se ilustra por el mismo ejemplo. En nuestro ejemplo, el acarreo se generará desde el dígito BCD inferior (el más a la derecha) al de más a la izquierda. El acarreo interno debe tenerse en cuenta y sumarlo al segundo dígito BCD. La instrucción de suma del 6502 se encarga de ello automáticamente. Sin embargo, a veces es conveniente detectar este acarreo interno desde el bit 3 al bit 4 (el "semiacarreo" o "acarreo intermedio"). El 6502 no tiene indicador de estado correspondiente.

Finalmente, tal como en el caso de la suma binaria, las instrucciones habituales SED y CLC se deben utilizar antes de ejecutar la suma BCD propiamente dicha. Como ejemplo se muestra, a continuación, un programa para sumar los números BCD "11" y "22":

```
CLC              BORRAR ACARREO
SED              PONER EN MODO DECIMAL
LDA    #$11      LITERAL BCD "11"
ADC    #$22      LITERAL BCD "22"
STA    ADR
```

En este programa, utilizamos dos nuevos símbolos: "#" y "$". El símbolo "#" significa que un "literal" (o constante) va a continuación. El signo

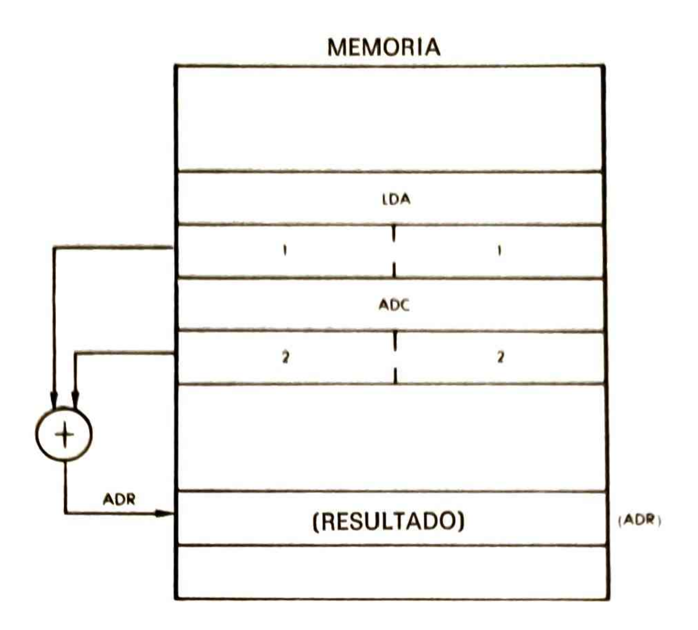


Figura 3-7 Almacenamiento de dígitos BCD.

"$" dentro del campo operando de la instrucción indica que los datos que siguen a continuación están expresados en notación hexadecimal. Las representaciones hexadecimales y BCD de dígitos "0" a "9" son idénticas. En este caso, deseamos sumar los literales (o constantes) "11" y "22". El resultado se almacena en la dirección ADR. Cuando el operando se indica como parte de la instrucción, como en el ejemplo anterior, se llama *direccionamiento inmediato.* (Los diferentes modos de direccionamiento se estudiarán en detalle en el capítulo 5.) El almacenamiento del resultado en una dirección específica, tal como STA ADR, se llama *direccionamiento absoluto* si ADR representa una dirección normal de 16 bits.

Ejercicio 3.6: *¿Podríamos desplazar la instrucción CLC en el programa por debajo de la instrucción LDA?*

Resta BCD

La resta BCD es compleja en apariencia. Para efectuar una resta BCD es preciso sumar el *complemento* a diez del número, de la misma manera que se suma el complemento a dos de un número para realizar una resta binaria. El complemento a diez se obtiene calculando el complemento a nueve y, después, sumándole 1. Ello suele exigir tres o cuatro operaciones en un microprocesador estándar. Sin embargo, el 6502 está provisto de una instrucción especial de resta BCD que realiza dicha operación en una sola instrucción. Naturalmente, y como en el ejemplo binario, el programa irá precedido por las instrucciones SED, que establecen el modo decimal, a menos que se haya

establecido previamente, y SEC que pone el acarreo a 1. Por tanto, el programa para restar el número BCD "25" del BCD "26" es el siguiente:

```
SED                  PONER MODO DECIMAL
SEC                  PONER A 1 ACARREO
LDA      #$26        CARGAR 26 EN BCD
SBC      #$25        MENOS 25 EN BCD
STA      ADR         ALMACENAR RESULTADO
```

Suma en BCD de 16 bits

La suma de 16 bits se efectúa tan simplemente como en el caso binario. El programa de dicha suma aparece a continuación:

```
CLC
SED
LDA      ADR1
ADC      ADR2
STA      ADR3
LDA      ADR1—1
ADC      ADR2—1
STA      ADR3—1
```

Ejercicio 3.7: *Comparar el programa anterior con el de la suma binaria de 16 bits. ¿Cuál es la diferencia?*

Ejercicio 3.8: *Escribir el programa de resta BCD en 16 bits. (No utilizar CLC ni ADC.)*

Indicadores BCD

En modo BCD, el indicador de acarreo durante una suma indica que el resultado es superior a 99. No es como en la situación de complemento a dos, ya que los dígitos BCD se representan en binario natural. Por el contrario, la ausencia de indicador de acarreo durante una resta indica un acarreo negativo.

Recomendaciones para la programación de sumas y restas

— Borrar siempre el indicador de acarreo antes de efectuar una suma.
— Poner siempre a uno el indicador de acarreo antes de efectuar una resta.
— Establecer el modo adecuado: binario o decimal.

Tipos de instrucción

Hemos utilizado hasta ahora tres tipos de instrucciones. Hemos empleado LDA y STA, que cargan el acumulador a partir de la dirección de memoria y almacenan su contenido en la dirección especificada. Estas dos instrucciones son de *transferencia de datos.*

A continuación hemos utilizado instrucciones *aritméticas,* tales como ADC y SBC. Ellas efectúan una suma y una resta respectivamente. En este capítulo se introducirán otras instrucciones de la ALU.

Finalmente, hemos utilizado instrucciones tales como CLC, SEC y otras, que manipulan los bits indicadores de estado (los bits de acarreo y decimal respectivamente en nuestros ejemplos). Son las instrucciones de *manipulación de estados* o de control. Una descripción completa de las instrucciones del 6502 se presentará en el capítulo 4.

Otros tipos de instrucciones también están disponibles en el microprocesador, las cuales no hemos utilizado aún. Son, en particular, las instrucciones de "bifurcación" y de "salto", que modificarán el orden de ejecución del programa. Este nuevo tipo de instrucciones se introducirá en nuestro próximo ejemplo.

Multiplicación

Examinemos ahora un problema aritmético más complejo: la multiplicación de números binarios. Con el fin de introducir el algoritmo de una multiplicación binaria, comencemos examinando una multiplicación decimal normal. Multiplicaremos 12 por 23.

12	(multiplicando)	(MPD)
× 23	(multiplicador)	(MPR)
36	(producto parcial)	(PP)
+ 24		
= 276	(resultado final)	(RES)

La multiplicación se efectúa realizando el producto del dígito más a la derecha del multiplicador por el multiplicando, o sea, "3" × "12". El producto parcial es "36". Después, se multiplica el dígito siguiente del multiplicador, o sea, "2" por "12". "24" se suma, entonces, al producto parcial.

Pero hay una operación más: 24 *se desplaza a la izquierda* una posición o, de modo equivalente, se podría decir que el producto parcial (36) *se ha desplazado una posición a la derecha* antes de la suma.

Los dos números, desplazados correctamente, se suman, entonces, y la suma es 276. Esto es sencillo. Consideremos ahora la multiplicación binaria. La multiplicación binaria se efectúa exactamente de igual modo.

Veamos un ejemplo. Multipliquemos 5 × 3:

(5)	101	(MPD)
(3)	× 011	(MPR)
	101	(PP)
	101	
	000	
(15)	01111	(RES)

Para efectuar la multiplicación operamos exactamente como hicimos anteriormente. La representación formal de este algoritmo aparece en la figura 3-8. Es el diagrama de flujo del algoritmo, nuestro primer diagrama de flujo. Examinémoslo más detenidamente.

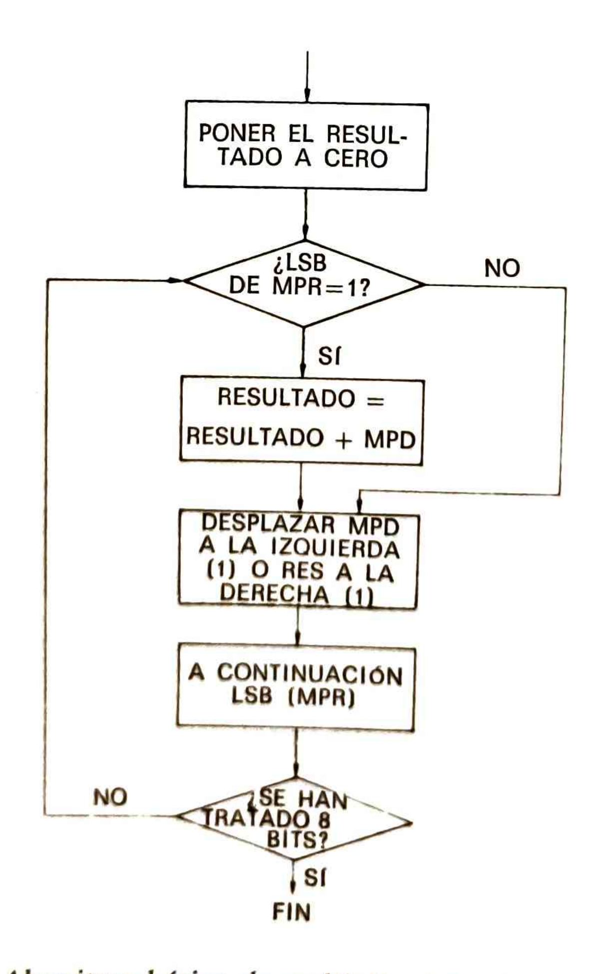


Figura 3-8 Algoritmo básico de multiplicación: diagrama de flujo.

Este diagrama de flujo es una representación simbólica del algoritmo que acabamos de presentar. Cada rectángulo representa una orden a ejecutar. Se traducirá en una o más instrucciones de programa. Cada rombo representa una comprobación. Corresponderá a un punto de bifurcación del programa. Si la comprobación resulta positiva, bifurcará a una posición determinada. Si la comprobación resulta negativa, bifurcará a otra posición. El concepto de bifurcación se explicará más tarde en el propio programa. El lector deberá examinar ahora este diagrama de flujo y averiguar que representa verdaderamente el algoritmo exacto. Obsérvese que hay una flecha que va desde el exterior del último rombo de la parte inferior del diagrama de flujo y que retorna al primer rombo de la parte superior. Esto es así porque la misma parte del diagrama de flujo se ejecutará ocho veces, una vez por cada bit del multiplicador. Tal situación en donde la ejecución vuelve a comenzar en el mismo punto se denomina *bucle o lazo de programa,* por razones obvias.

Ejercicio 3.9: *Multiplique "4" por "7" en binario utilizando el diagrama de flujo y compruebe que se obtiene "28". En caso contrario, inténtelo de nuevo. Solamente si obtiene el resultado correcto, está preparado para convertir este diagrama de flujo en un programa.*

Convirtamos, ahora, este diagrama de flujo en un programa para el 6502. El programa completo aparece en la figura 3-9. A continuación, lo vamos a

```
        LDA   #0        ACUMULADOR A CERO
        STA   TMP       BORRAR ESTA DIRECCIÓN
        STA   RESAD     BORRAR
        STA   RESAD+1   BORRAR
        LDX   #8        X ES UN CONTADOR
MULT    LSR   MPRAD     DESPLAZAR MPR A LA DERECHA
        BCC   NO ADD    COMPROBAR EL BIT DE ACARREO
        LDA   RESAD     CARGAR A CON PARTE BAJA DE RES
        CLC             PREPARAR PARA SUMAR
        ADC   MPDAD     SUMAR MPD A RES
        STA   RESAD     CONSERVAR RESULTADO
        LDA   RESAD+1   SUMAR EL RESTO DE MPD DESPLAZADO
        ADC   TMP
        STA   RESAD+1
NOADD   ASL   MPDAD     DESPLAZAR MPD A LA IZQUIERDA
        ROL   TMP       CONSERVAR BIT DE MPD
        DEX             DECREMENTAR CONTADOR
        BNE   MULT      VOLVER A COMENZAR SI CONTADOR ≠ 0
```

Figura 3-9 Multiplicación 8 por 8.

estudiar en detalle. Como recordará del capítulo 1, la programación consiste, en este caso, en la conversión del diagrama de flujo de la figura 3-8 en el programa de la figura 3-9. Cada uno de los bloques del diagrama de flujo se convertirá en una o más instrucciones.

Se supone que MPR y MPD tienen ya un valor.

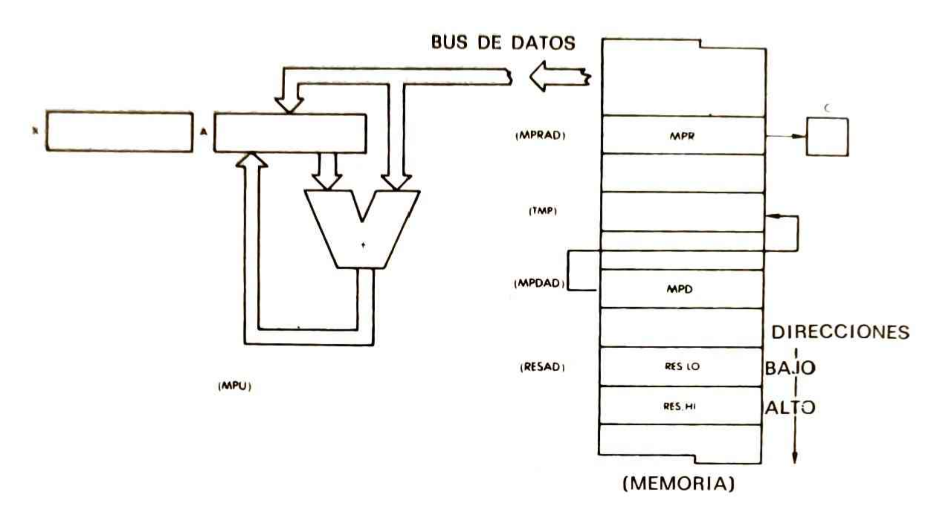


Figura 3-10 Multiplicación: los registros.

El primer bloque del diagrama de flujo es un bloque de *inicialización.* Es necesario poner a "0" un cierto número de registros, o posiciones de memoria, ya que este programa requerirá su empleo. Los registros que se utilizarán por el programa de multiplicación aparecen en la figura 3-10. A la izquierda de la ilustración aparecen las partes más importantes del microprocesador 6502. A la derecha de la ilustración aparece la sección destacada de la memoria. Supondremos que las direcciones de memoria aumentan desde la parte superior a la inferior de la ilustración. Naturalmente, se podrá utilizar la notación inversa. El registro X en la parte más a la izquierda (uno de los registros índice del 6502) se utilizará como *contador.* Ya que estamos haciendo una multiplicación de 8 bits, habremos de comprobar los 8 bits del multiplicador. Lamentablemente no hay ninguna instrucción en el 6502 que nos permita comprobar esos bits en secuencia. Los únicos bits que pueden ser comprobados convenientemente son los indicadores en el registro de estados. Como resultado de esta limitación común a la mayoría de los microprocesadores, para comprobar todos los bits del multiplicador sucesivamente será necesario transferir el valor del multiplicador al acumulador. A continuación se desplazará a la derecha el contenido del acumulador. Una instrucción de desplazamiento desplaza cada bit de registro una posición a la derecha o a la izquierda. El bit que sale del registro va al bit

de acarreo del registro de estado. El efecto de una operación de desplazamiento se ilustra en la figura 3-11. Existen muchas variaciones posibles que dependen del bit que se introduce en el registro, pero estas diferencias se comentarán en el capítulo 4.

Volvamos a la comprobación sucesiva de cada uno de los 8 bits del multiplicador. Ya que se puede comprobar fácilmente el bit de acarreo, el multiplicador se desplazará en una posición ocho veces. En cada ocasión, su bit más a la derecha irá el bit de acarreo, en donde se comprobará.

El siguiente problema a resolver es que el producto parcial que se acumula durante las sumas sucesivas requerirá 16 bits. La multiplicación de números de 8 bits puede producir un resultado de 16 bits. Esto es debido a que $2^8 \times 2^8 = 2^{16}$. Necesitamos, pues, reservar 16 bits para el resultado. Lamentablemente el 6502 tiene muy pocos registros internos, de modo que este producto parcial no se puede almacenar dentro del 6502 propiamente dicho. De hecho, debido al número limitado de registros, no podemos almacenar el multiplicador, el multiplicando, o el producto parcial en el interior del 6502. Se almacenarán todos en la memoria. Resultará, por tanto, una ejecución más lenta que si fuera posible almacenarlos todos en registros internos. Esta es una limitación inherente al 6502. La zona de memoria utilizada para la multiplicación aparece a la derecha de la figura 3-10. En la parte superior se puede ver la palabra de memoria asignada al multiplicador. Supondremos, por ejemplo, que contiene "3" en binario. La dirección de esta posición de memoria es MPRAD. Más adelante encontraremos una "zona temporal" cuya dirección es TMP. La misión de esta posi-

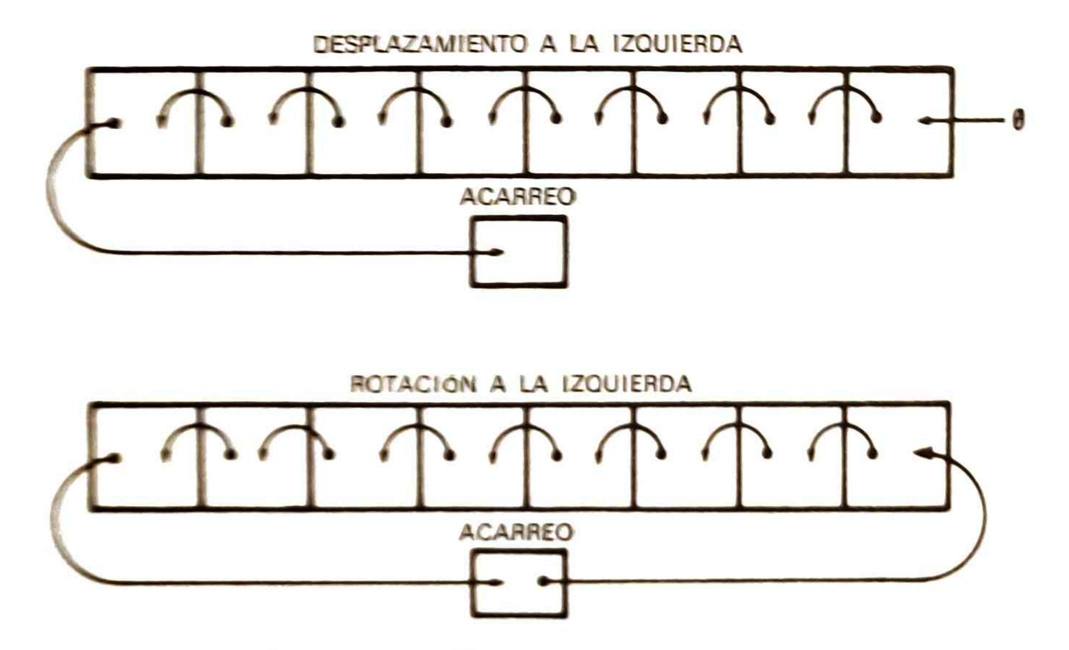


Figura 3-11 Desplazamiento y rotación.

ción se aclarará a continuación. Desplazaremos el multiplicando a la izquierda a su posición, antes de sumarlo al producto parcial. El multiplicando está a continuación y se supondrá contiene el valor "5" en binario. Su dirección es MPDAD.

Finalmente, en la parte inferior de la memoria encontraremos las dos palabras asignadas al producto parcial o resultado. Su dirección es RESAD.

Estas posiciones de memoria serán nuestros "registros de trabajo" y el "registro" de palabras se puede utilizar de modo intercambiable con la "posición de memoria" en este contexto.

La flecha que aparece en la parte superior derecha de la ilustración y que va de MPR al bit C, es un medio simbólico de mostrar cómo se desplazará el multiplicador en el bit de acarreo. Naturalmente, este bit de acarreo está contenido físicamente en el interior del 6502 y no en el interior de la memoria.

Volvamos de nuevo al programa de la figura 3-9. Las primeras cinco instrucciones son del tipo de inicialización:

Las primeras cuatro instrucciones borrarán los contenidos de los "registros" TMP, RESAD y RESAD+1. Verifiquemos esta circunstancia.

LDA # 0

Esta instrucción carga el acumulador con el valor literal "0". Resulta de ello que el acumulador contendrá "00000000".

El contenido del acumulador se utilizará ahora para borrar los tres "registros" en la memoria. Es preciso recordar que la lectura de un registro no lo vacía. Es posible leer de un registro cuantas veces sea necesario. Su contenido no se modifica por la operación de lectura. Continuemos:

STA TMP

Esta instrucción almacena el contenido del acumulador en la posición de memoria TMP. Véase la figura 3-10 para comprender el flujo de datos en el sistema. El acumulador contiene "00000000". El resultado de esta instrucción será poner a cero toda la posición de memoria TMP. Recuerde que el contenido del acumulador queda en "0" después de una operación de lectura del mismo. Es inalterable. Vamos a utilizarlo de nuevo.

STA RESAD

Esta instrucción actúa como la precedente y borra el contenido de la dirección RESAD. Hagámoslo una vez más:

STA RESAD + 1

Borremos finalmente la posición de memoria RESAD+1 que se ha reservado para almacenar la parte alta del resultado. (La mitad alta son los bits 8-15; la parte baja son los bits 0-7.)

Por último, para poder detener el desplazamiento de los bits del multiplicador en el momento adecuado, es necesario contar el número de desplazamientos que se han realizado. Ocho desplazamientos son necesarios. El registro X se utilizará como contador y se inicializa al valor "8". Cada vez que se haya efectuado un desplazamiento, el contenido de este contador se decrementará en 1. Siempre que el valor del contador alcance "0", se termina la multiplicación. Inicialicemos este registro a "8".

```
LDX # 8
```

Esta instrucción carga la constante "8" en el registro X.

Con referencia, de nuevo, al diagrama de flujo de la figura 3-8, debemos comprobar el bit menos significativo del multiplicador. Se ha indicado anteriormente que esta prueba no puede ser realizada por una sola instrucción. Se deben utilizar dos instrucciones. En primer lugar, el multiplicador se desplazará a la derecha y después se comprobará el bit que se extrae. Es el bit de acarreo. Efectuemos estas operaciones:

```
LSR MPRAD
```

Esta instrucción es un "desplazamiento lógico a derecha" del contenido en la posición de memoria MPRAD.

Ejercicio 3.10: *Supongamos que el multiplicador es "3" en nuestro ejemplo, ¿Cuál es el bit que sale por la derecha de la posición de memoria MPRAD? (Dicho de otro modo, ¿cuál será el valor del acarreo después de este desplazamiento?)*

La instrucción siguiente comprueba el valor del bit de acarreo:

```
BCC NOADD
```

Esta instrucción significa "bifurcación si el acarreo es cero" en la dirección NOADD.

Es la primera vez que encontramos una instrucción de bifurcación. Todos los programas que hemos considerado hasta aquí han sido estrictamente secuenciales. Cada instrucción se ejecutó después de la anterior. Para poder utilizar pruebas lógicas tales como la comprobación del bit de acarreo, se deben poder ejecutar instrucciones en cualquier parte del programa después

de la comprobación. La instrucción de bifurcación se efectúa como una función. Se comprobará el valor del bit de acarreo. Si el acarreo fuera "0", esto es, si se hubiera borrado, entonces el programa bifurcará a la dirección NOADD. Esto significa que la siguiente instrucción ejecutada después de BCC será la instrucción en la dirección NOADD si la comprobación fue positiva.

En caso contrario, si la comprobación resulta negativa, no se producirá ninguna bifurcación y la instrucción siguiente se ejecutará normalmente.

Un nuevo significado se da al término NOADD: se trata de una *etiqueta simbólica*. Representa una dirección física real en el interior de la memoria. Para comodidad del programador, el programa ensamblador permite utilizar nombres simbólicos en lugar de direcciones reales. Durante el proceso de ensamblado, el ensamblador sustituirá el símbolo "NOADD" por la dirección física real. Esto mejora notablemente la legibilidad del programa y permite también al programador insertar instrucciones adicionales entre el punto de bifurcación y NOADD, sin tener que volverlo a escribir por completo. Estas ventajas se estudiarán con más detalle en el capítulo 10 en el apartado del ensamblador.

Si la prueba es negativa, se ejecuta la siguiente instrucción secuencial en el programa. Estudiaremos ambas alternativas:

Alternativa 1: el acarreo fue "1"

Si el acarreo fue 1, la prueba indicada por BCC es negativa y se ejecuta la siguiente instrucción después de BCC.

LDA RESAD

Alternativa 2: el acarreo fue "0"

La prueba resultó positiva y la instrucción siguiente es la de etiqueta "NOADD".

Consultando la figura 3-8, el diagrama de flujo especifica que si el bit de acarreo fue 1, el multiplicando se debe sumar al producto parcial (aquí los registros RES). También se debe efectuar un desplazamiento. El producto parcial se debe desplazar una posición a la derecha o, de no ser así, el multiplicando se debe desplazar una posición a la izquierda. Adoptaremos aquí la notación habitual que se utiliza cuando se realiza la multiplicación a mano y desplazaremos el multiplicando una posición a la izquierda.

El multiplicando está contenido en los registros TMP y MPDAD. (Para simplificar, llamaremos a las posiciones de memoria "registros" como término habitual.) Los 16 bits del producto parcial están contenidos en las direcciones de memoria RESAD y RESAD+1.

Con el fin de ilustrar esto último, supongamos que el multiplicando es "5". Los diversos registros aparecen en la figura 3-10.

Tenemos que sumar simplemente dos números de 16 bits. Se trata de un problema que hemos aprendido a solucionar. (Si se tiene cualquier duda, consúltese la sección anterior de suma de 16 bits.) Vamos, en primer lugar, a sumar los bytes de orden (peso) bajo (menos significativos) y después los bytes de orden alto (más significativos). Hagámoslo:

```
LDA RESAD
```

El acumulador se carga con la parte baja de RES.

```
CLC
```

Antes de cualquier suma, el 6502 requiere que se borre el bit de acarreo. Es importante hacerlo ahora ya que sabemos que el bit de acarreo había sido puesto a 1. Debe ser borrado.

```
ADC MPDAD
```

El multiplicando se suma al acumulador, que contiene la parte baja de RES (RES LOW).

```
STA RESAD
```

El resultado de la suma se conserva en la posición adecuada de memoria, (RES)LOW. Entonces se realiza la segunda mitad de la suma. Cuando más adelante compruebe a mano la ejecución de este programa, no olvide que la suma posicionará el bit de acarreo. El acarreo se pondrá a "0" o a "1" según los resultados de la suma. Cualquier acarreo que pueda haber sido generado se llevará automáticamente hacia la parte de orden alta del resultado.

Terminemos, ahora, la suma:

```
LDA    RESAD+1
ADC    TMP
STA    RESAD+1
```

Estas tres instrucciones completan nuestra suma de 16 bits. Hemos sumado el multiplicando a RES. Tenemos que desplazarlo una posición a la izquierda con miras a la próxima suma. Podríamos haber considerado también el desplazamiento del multiplicando una posición a la izquierda *antes* de la

suma, salvo la primera vez. Esta es una de las muchas opciones de programación que están siempre abiertas al programador.

Desplacemos el multiplicando a la izquierda:

NOADD ASL MPDAD

Esta instrucción es un "desplazamiento aritmético a la izquierda". Desplazará una posición a la izquierda el contenido de la posición de memoria MPDAD, que contiene la parte baja del multiplicando. Esto no es suficiente. No podemos permitirnos perder el bit que sale por la parte izquierda del multiplicando. Este bit irá al bit de acarreo. No debe ser almacenado allí permanentemente, ya que puede ser destruido por cualquier operación aritmética. Este bit se debe conservar en un registro "permanente". Se deberá desplazar a la posición de memoria TMP.
Es precisamente realizado por la instrucción siguiente:

ROL TMP

Esta especifica: "desplazamiento a la izquierda" del contenido de TMP.

Se debe hacer una observación importante en este punto. Acabamos de utilizar dos clases diferentes de instrucciones de desplazamiento para desplazar un registro en una posición a la izquierda. La primera es ASL. La segunda es ROL. ¿Cuál es la diferencia?

La instrucción ASL desplaza el contenido del registro. La instrucción ROL es una instrucción de rotación o giro. Desplaza el contenido del registro una posición a la izquierda y el bit que sale por la izquierda va al bit de acarreo, como es habitual. La diferencia es que *el contenido anterior del bit de acarreo se introduce en la posición más a la derecha*. En matemáticas se llama a esta operación una rotación circular (una rotación de 9 bits). Es exactamente lo que deseamos. Como resultado de ROL, el bit que se ha desplazado fuera de MPDAD por la izquierda y que estaba conservado en el bit de acarreo irá a parar a la posición más a la derecha del registro TMP. Esta es la forma adecuada de funcionar.

Hemos terminado con la parte aritmética de este programa. Tenemos que comprobar todavía si hemos realizado la operación ocho veces, o sea, si realmente hemos terminado. Como es habitual en la mayoría de los microprocesadores, esta comprobación requiere dos instrucciones:

DEX

Esta instrucción decrementa el contenido del registro X. Si contiene 8, su contenido será 7 después de la ejecución de esta instrucción.

BNE MULT

Esta es otra instrucción de prueba y bifurcación. Significa que "bifurque a la posición MULT si el resultado no es igual a 0". Mientras nuestro registro contador se decrementa hasta un entero distinto de cero, se bifurcará de nuevo automáticamente a la dirección de etiqueta MULT. A esto se denomina bucle de multiplicación. Con referencia al diagrama de flujo de la multiplicación, esto corresponde a la flecha que sale del último bloque. Este bucle se ejecutará 8 veces.

Ejercicio 3.11: *¿Qué sucede si X disminuye hasta 0? ¿Cuál es la siguiente instrucción que se ejecuta?*

En la mayoría de los casos el programa que se acaba de desarrollar será una subrutina y la instrucción final de la subrutina será RTS. El mecanismo de la subrutina se explicará más adelante en este capítulo.

AUTOCOMPROBACIÓN IMPORTANTE

Si se desea aprender cómo programar, es extremadamente importante asimilar un programa típico de este género, en todos sus detalles. Hemos introducido muchas instrucciones nuevas. El algoritmo es razonablemente sencillo, pero el programa es mucho más largo que los que se han desarrollado hasta ahora. *Se le recomienda muy encarecidamente que haga el ejercicio siguiente completa y correctamente antes de continuar con este capítulo.* Si lo hace correctamente, habrá comprendido realmente el mecanismo por el que las instrucciones manipulan el contenido de la memoria y de los registros del microprocesador y cómo se utiliza el indicador de acarreo. Si no lo hace, es probable que tendrá dificultades cuando escriba los programas por su propia cuenta. El aprendizaje de la programación necesita que programe usted mismo. Sírvase tomar una hoja de papel y haga el ejercicio siguiente:

Ejercicio 3.12: *Cada vez que escribe un programa, se debe verificar a mano y con el fin de cerciorarse de que los resultados son correctos. Vamos a hacer precisamente eso: el objetivo de este ejercicio es rellenar la tabla de la figura 3-12.*

Puede escribir directamente en ella o bien hacer una copia de la misma. El propósito es determinar el contenido de cada registro significativo y la posición de memoria en el sistema, después de que cada instrucción del programa se haya ejecutado desde el principio al final. Encontrará sobre el encabezamiento horizontal de la figura 3-12 todas las posiciones de los registros utilizados por el programa: X, A, MPR, C (indicador del bit de aca-

ETIQUETA	INSTRUCCIÓN	X	A	MPR	C	TEMP	MPD	(RESAD)L	(RESAD)H

Figura 3-12 Tabla a rellenar para el ejercicio 3.12.

rreo), TMP, MPD, RESADL, RESADH. En la parte izquierda de la ilustración se debe poner la etiqueta, si es aplicable, y la instrucción que se está ejecutando. A la derecha de la ilustración se debe escribir el contenido de cada registro después de la ejecución de esta instrucción. Siempre que el contenido de un registro sea indefinido, se utilizan rayas. Comencemos a rellenar esta tabla. Tendrá que rellenar el resto por sí solo. La primera línea aparece a continuación:

ETIQUETA	INSTRUCCIÓN	X	A	MPR	C	TEMP	MPD	(RESAD)L	(RESAD)H
	LDA #0	------	00000000	00000011	--	------	00000101	------	------

Figura 3-13 Primera instrucción de la multiplicación.

La primera instrucción que se ejecuta es LDA # 0.

Después de la ejecución de esta instrucción, el contenido del registro X es desconocido. Éste se indica por rayas. El contenido del acumulador es todo ceros. Supondremos también que el multiplicador y el multiplicando han sido creados por el programador antes de la ejecución de este programa. (En caso contrario, se necesitarán instrucciones adicionales para establecer los contenidos de MPR y MPD.) Encontramos en MPR el valor binario de "3" y en MPD el valor binario de "5". El bit de acarreo no está definido. El registro TMP y los dos registros utilizados por RESAD no están definidos. Rellenemos ahora la línea siguiente. Es patente que la única diferencia es que el contenido del registro TMP se ha puesto a "0". La siguiente instrucción pondrá el contenido de RESAD+1 también a "0".

ETIQUETA	INSTRUCCIÓN	X	A	MPR	C	TEMP	MPD	(RESAD)L	(RESAD)H
	LDA #0	------	00000000	00000011	--	------	00000101	------	------
	STA TEMP					00000000			

Figura 3-14 Las dos primeras líneas de multiplicación.

La quinta instrucción: LDX#8, pondrá el contenido de X a "8". Examinemos un juego de instrucciones más (figura 3-15).

La instrucción LSR MPRAD desplazará una posición a la derecha el contenido de MPRAD. Puede constatar que, después del desplazamiento, el contenido de MPR es "0000 0001". El "1" más a la derecha de MPR ha ido al bit de acarreo. El bit C se pone ahora a 1. Los otros registros no cambian.

Ahora sírvase completar el resto de esta tabla. No es difícil, pero requiere atención. Si tiene dudas acerca de la función de algunas instrucciones, puede

ETIQUETA	INSTRUCCIÓN	X	A	MPR	C	TEMP	MPD	(RESAD)L	(RESAD)H
	LDA #0	------	00000000	00000011	--	------	00000101	------	------
	STA TEMP					00000000			
000	STA RESAD							00000000	
	STA RESAD + 1								00000000
	LDX #8	00001000							
MULT	LSR MPRAD			00000001	1				
	BCC NOADD								
	LDA RESAD								
	CLC				0				
	ADC MPDAD		00000101						
101	STA RESAD							00000101	
	LDA RESAD + 1		00000000						
	ADC TEMP								
	STA RESAD + 1								
NOADD	ASL MPDAD						00001010		
	ROL TEMP								
	DEX	00000111							
	BNE MULT								
MULT	LSR MPRAD			00000000	1				
	2.ª ITERACIÓN								

Figura 3-15 Tabla del ejercicio 3.12 rellenada parcialmente.

consultar el capítulo 4, en donde encontrará descritas cada una de ellas, o bien la sección de apéndices de este libro donde están listadas en forma de tablas.

El resultado final de su multiplicación debe ser "15" en forma binaria, contenido en los registros RESAD bajo y alto. RESAD alto debe ser puesto a "0000 0000". RESAD bajo debe ser "0000 1111". Si obtiene este resultado ha triunfado. Si no fuera así, inténtelo una vez más. La fuente más frecuente de errores es el deficiente tratamiento del bit de acarreo. Asegúrese de que el bit de acarreo se cambia cada vez que se realiza una instrucción aritmética. No olvide que la ALU establecerá el bit de acarreo después de cada operación de suma.

Alternativas de programación

El programa que acabamos de desarrollar es una de las muchas formas en que puede haber sido escrito. Cada programador puede encontrar medios de modificar y mejorar, en ocasiones, un programa. Por ejemplo, hemos desplazado el multiplicando a la izquierda antes de la suma. Habría sido matemáticamente equivalente desplazar el resultado una posición a la derecha antes de sumarlo al multiplicando. La ventaja es que no se haya requerido el registro TMP, lo que ahorra una posición de memoria. Este método se preferirá en un microprocesador provisto de bastantes registros internos de modo que MPR, MPD y RESAD podrán estar contenidos en el interior del microprocesador. Ya que estamos obligados a utilizar la memoria para realizar estas operaciones, el ahorro de una posición de memoria no tiene importancia. La cuestión es, por tanto, saber si con el segundo método puede resultar la multiplicación algo más rápida. Este es un ejercicio interesante:

Ejercicio 3.13: *Escriba una multiplicación 8×8, utilizando el mismo algoritmo, pero desplazando el resultado una posición a la derecha en lugar de desplazar el multiplicando una posición a la izquierda. Compárelo con el programa anterior y determine si esta versión diferente puede ser más rápida o más lenta que la precedente.*

Un problema más puede plantearse: Para determinar la velocidad del programa, puede necesitar consultar algunas de las tablas de los apéndices que listan el número de ciclos requeridos para cada instrucción. Sin embargo, el número de ciclos requeridos por algunas instrucciones depende de dónde están situados. El 6502 tiene un modo especial de direccionamiento denominado modo de *direccionamiento directo de página cero,* en donde la primera página (posiciones de memoria 0 a 255) se reserva para ejecuciones rápidas. Esto se explicará en el capítulo 5 que trata de las técnicas de direccionamiento. Dicho en pocas palabras, todos los programas que requieren un tiempo de ejecución breve utilizarán variables situadas en la página 0, de modo que las instrucciones solamente requieren dos bytes para direccionar las posiciones de memoria (el direccionamiento de 256 posiciones sólo precisa un byte), mientras que las instrucciones localizadas en cualquier otra parte de la memoria requerirán normalmente 3 bytes. Aplacemos este análisis hasta el capítulo 5.

Programa perfeccionado de multiplicación

El programa que acabamos de desarrollar es una traducción evidente del algoritmo en código. Sin embargo, la programación eficaz requiere una atención más completa para detallar de qué modo se puede reducir la longitud del programa y mejorar su velocidad de ejecución. Vamos a presentar una realización mejorada del mismo algoritmo.

Una de las tareas que consumen instrucciones y tiempo es el desplazamiento del resultado y del multiplicador. Un "artificio" normal utilizado en el algoritmo de la multiplicación está basado en la observación siguiente: cada vez que el multiplicador se desplaza la posición de un bit a la derecha, queda disponible a la izquierda una posición de un bit. Simultáneamente se puede observar que el primer resultado (o producto parcial) utilizará todo lo más 9 bits. Después del desplazamiento de la siguiente multiplicación, la magnitud del producto parcial aumentará, de nuevo, en un bit. En otras palabras, podemos reservar inicialmente una posición de memoria para el producto parcial y luego utilizar las posiciones de los bits que se están liberando por el multiplicador mientras que se desplaza.

Vamos a desplazar el multiplicador a la derecha. Dejará libre una posición de un bit a la izquierda. Vamos a introducir el bit más a la derecha

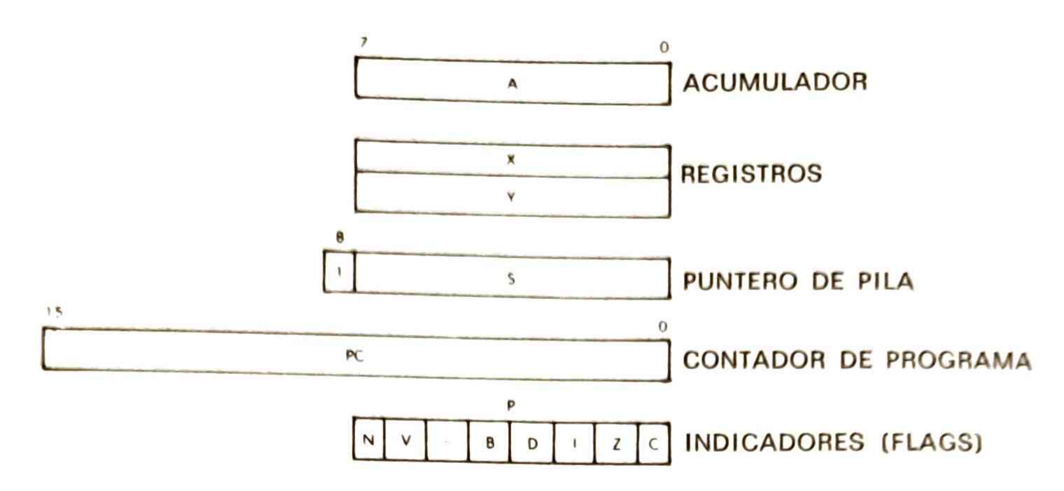


Figura 3-16 Registros del 6502.

del producto parcial en la posición del bit que se ha quedado libre. Consideremos ahora el programa.

Consideremos también el empleo óptimo de los registros. Los registros internos del 6502 aparecen en la figura 3-16. Lo mejor es utilizar X como contador. Lo utilizaremos para contar el número de bits desplazados. El acumulador es (desgraciadamente) el único registro interno que se puede desplazar. Para mejorar la eficacia almacenaremos en el mismo el multiplicador o el resultado.

¿Qué debemos poner en el acumulador? El resultado se debe sumar al multiplicando cada vez que 1 se desplace fuera. Ya que el 6502 suma siempre algo solamente al acumulador, es el resultado lo que residirá en el acumulador.

Los otros números tendrán que residir en la memoria (ver figura 3-17).

A y B contendrán el resultado. A contendrá la parte alta del resultado y B contendrá la parte baja del mismo. A es el acumulador y B es una posición

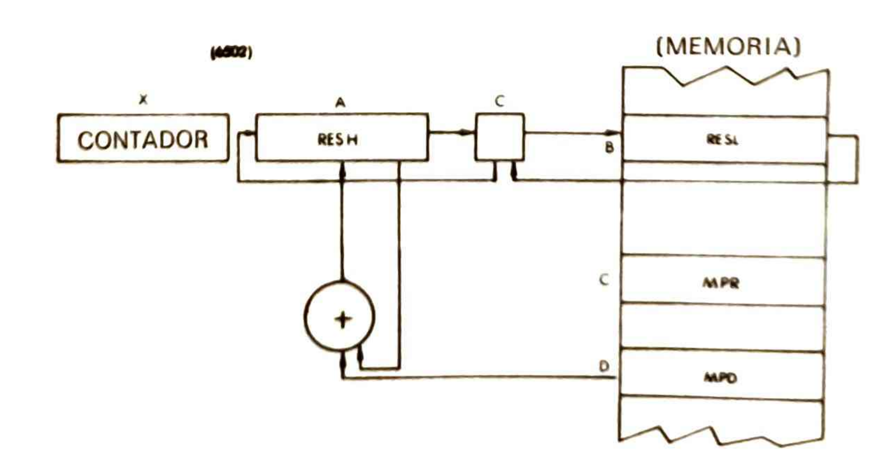


Figura 3-17 Asignación de registros (multiplicación perfeccionada).

de memoria, preferiblemente en la página 0. C contendrá el multiplicador (una posición de memoria). D contiene el multiplicando (una posición de memoria). El programa correspondiente es el siguiente:

```
MULT    LDA   #0      INICIALIZAR RESULTADO A CERO (ALTO)
        STA   B       INICIALIZAR RESULTADO (BAJO)
        LDX   #8      X ES EL CONTADOR DE DESPLAZAMIENTO
LOOP    LSR   C       DESPLAZAR MPR
        BCC   NOADD
        CLC           EL ACARREO FUE UNO. BORRARLO
        ADC   D       A = A + MPD
NOADD   ROR   A       DESPLAZAR RESULTADO
        ROR   B       RECUPERAR BIT EN B
        DEX           DECREMENTAR EL CONTADOR
        BNE   LOOP    ¿ÚLTIMO DESPLAZAMIENTO?
```

Figura 3-18 Multiplicación perfeccionada.

Examinemos el programa. Ya que A y B contendrán el resultado, se deben inicializar al valor 0. Hagámoslo:

```
MULT LDA#0
     STA B
```

En este caso utilizaremos el registro X como un contador de desplazamiento y se inicializará al valor 8:

```
LDX#8
```

Ahora estamos preparados para introducir el bucle principal de multiplicación como antes. Desplazamos, en primer lugar, el multiplicador y después se comprueba el bit de acarreo que contiene el bit más a la derecha del multiplicador, que ha quedado fuera. Hagámoslo:

```
LOOP LSR C
     BCC NOADD
```

Desplazamos el multiplicador a la derecha como antes. Esto es equivalente al algoritmo anterior, ya que la operación de suma se dice que es expansiva.

Existen dos posibilidades: si el acarreo era 0, bifurcaremos a NOADD. Supongamos que el acarreo es 1. Continuemos:

```
CLC
ADC D
```

Ya que el acarreo fue 1, se debe borrar y sumaremos el multiplicando al acumulador. (El acumulador contiene el resultado, 0 hasta este momento.) Desplacemos ahora el producto parcial:

```
NOADD ROR A
      ROR B
```

El producto parcial en A se desplaza a la derecha un bit. El bit más a la derecha va al bit de acarreo. El bit de acarreo se atrapa y se efectúa una rotación en el registro B, que contiene la parte baja del resultado.

Hemos de comprobar simplemente si hemos terminado:

```
DEX
BNE LOOP
```

Si examinamos este nuevo programa, vemos que se ha escrito en aproximadamente la mitad del número de instrucciones del programa anterior. Se ejecutará también mucho más rápidamente. Ello pone de manifiesto la utilidad de seleccionar los registros adecuados que contengan la información.

Una escritura de programa directa dará lugar a un programa que funciona correctamente, pero no proporcionará un programa que esté optimizado. Es, por tanto, de considerable importancia utilizar los registros disponibles y las posiciones de memoria del mejor modo posible. Este ejemplo ilustra una aproximación racional en la elección de registro para conseguir una eficacia máxima.

Ejercicio 3.14: *Calcular la velocidad de una operación de multiplicación utilizando este último programa. Supóngase que una bifurcación ocurrirá en el cincuenta por ciento de los casos. Obsérvese el número de ciclos requeridos por cada instrucción en la tabla de final del libro. Supóngase una frecuencia de reloj de un ciclo = 1 microsegundo.*

División binaria

El algoritmo de la división binaria es análogo al que se ha utilizado para la multiplicación. El divisor se resta sucesivamente de los bits de orden elevado del dividendo. Después de cada resta, se utiliza el resultado en lugar del dividendo inicial. El valor del cociente se incrementa simultáneamente en 1 cada vez. Eventualmente, el resultado de la resta es negativo. A ello se llama un rebasamiento. Se debe restablecer entonces el resultado parcial sumándole de nuevo el divisor. Naturalmente, el cociente se debe disminuir

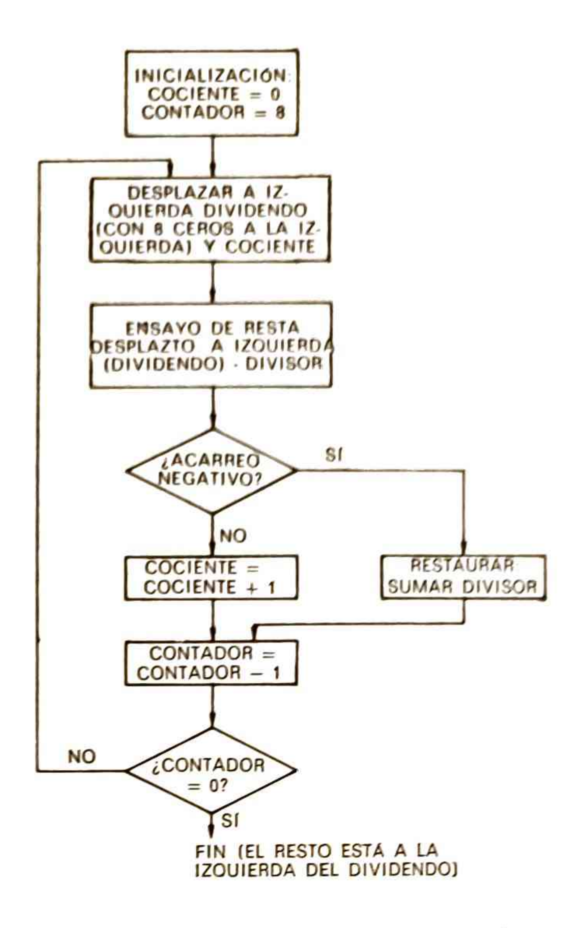


Figura 3-19 Diagrama de flujo de la división binaria de 8 bits.

en 1 simultáneamente. El cociente y el dividendo se desplazan después una posición de un bit a la izquierda y se repite el algoritmo.

El método que acabamos de describir se denomina *método con restablecimiento.* Una variante de este método que produce una mejora en la velocidad de ejecución se llama *método sin restablecimiento.*

La división de 16 bits

Ahora se describirá la división sin restablecimiento de un dividendo de 16 bits y un divisor de 8 bits. El resultado tendrá 8 bits. El registro y la posición de memoria de este programa se muestran en la figura 3-22. El dividendo está contenido en el acumulador (parte alta) y en la posición de memoria 0, llamada aquí B. El resultado está contenido en Q (posición de memoria 1). El divisor está contenido en D (posición de memoria 2). El resultado estará contenido en Q y A (A contendrá el resto).

El programa aparece en la figura 3-21 y el diagrama de flujo correspondiente se muestra en la figura 3-20.

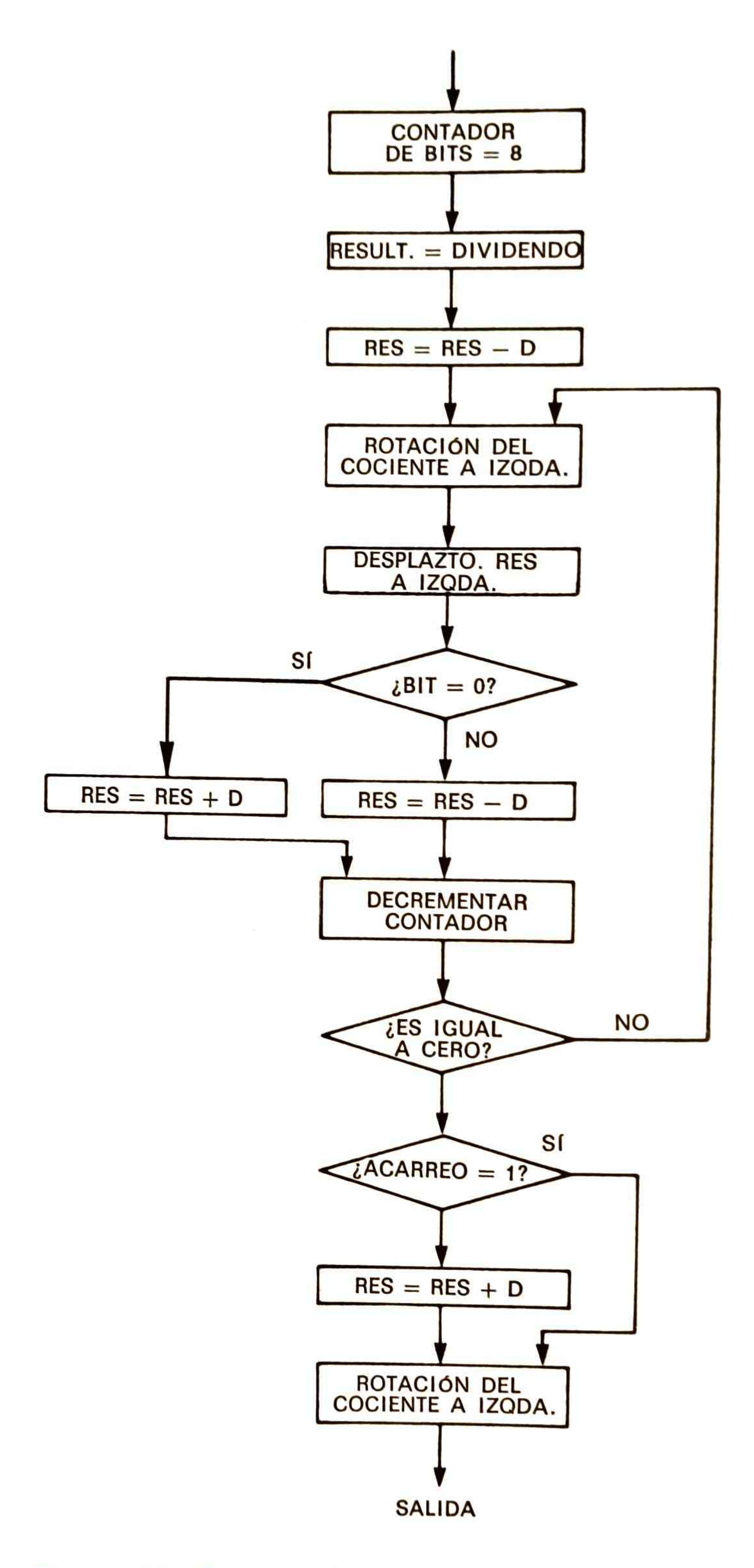


Figura 3-20 Diagrama de flujo de la división 16 por 8.

```
LINE #  LOC     CODE        LINE

0002    0000                        * = $0
0003    0000                B       * = * + 1
0004    0001                Q       * = * + 1
0005    0002                D       * = * + 1
0006    0003                        * = $200
0007    0200    A0 08       DIV     LDY #8
0008    0202    38                  SEC
0009    0203    E5 02               SBC D
0010    0205    08          LOOP    PHP
0011    0206    26 01               ROL Q
0012    0208    06 00               ASL B
0013    020A    2A                  ROL A
0014    020B    28                  PLP
0015    020C    90 05               BCC ADD
0016    020E    E5 02               SBC D
0017    0210    4C 15 02            JMP NEXT
0018    0213    65 02       ADD     ADC D
0019    0215    88          NEXT    DEY
0020    0216    D0 ED               BNE LOOP
0021    0218    B0 03               BCS LAST
0022    021A    65 02               ADC D
0023    021C    18                  CLC
0024    021D    26 01       LAST    ROL Q
0025    021F    00                  BRK
0026    0220                        END
```

Figura 3-21 Programa de la división 16 por 8.

Ejercicio 3.15: *Verifique el funcionamiento correcto de este programa ejecutando la división a mano y haga como ejercicio el programa que se hizo en el ejercicio 3.12. Divida 33 por 3. El resultado debe ser, naturalmente, 11, con un resto de 0.*

OPERACIONES LÓGICAS

Las otras clases de instrucciones que puede ejecutar la ALU en el interior del microprocesador son el juego de instrucciones lógicas. Ellas incluyen AND (Y), OR (O) y exclusive OR (OR esclusiva) (EOR). Además, pueden también incluirse en las mismas las operaciones de desplazamiento que han sido ya utilizadas y la instrucción de comparación, llamada CMP en el 6502. El empleo individual de AND, ORA, EOR se describirá en el capítulo 4 en el juego de instrucciones del 6502. Desarrollemos ahora un programa corto que comprobará si una posición de memoria dada, llamada LOC, contiene el valor "0", el valor "1" o cualquier otro. El programa se muestra a continuación:

```
                  LDA    LOC     LEER EL CARÁCTER EN LOC
                  CMP    #$00    COMPARAR CON CERO
                  BEQ    CERO    ¿ES 0?
                  CMP    #$01    ¿ES 1?
                  BEC    UNO
NINGUNO
  ENCONTRADO      ...
CERO              ...
UNO               ...
```

La primera instrucción: LDA LOC lee el contenido de la posición de memoria LOC. Este es el carácter que deseamos comprobar.

CMP #$00

Esta instrucción compara el contenido del acumulador con la constante hexadecimal del valor "00" (o sea, la configuración de bits "0000 0000").

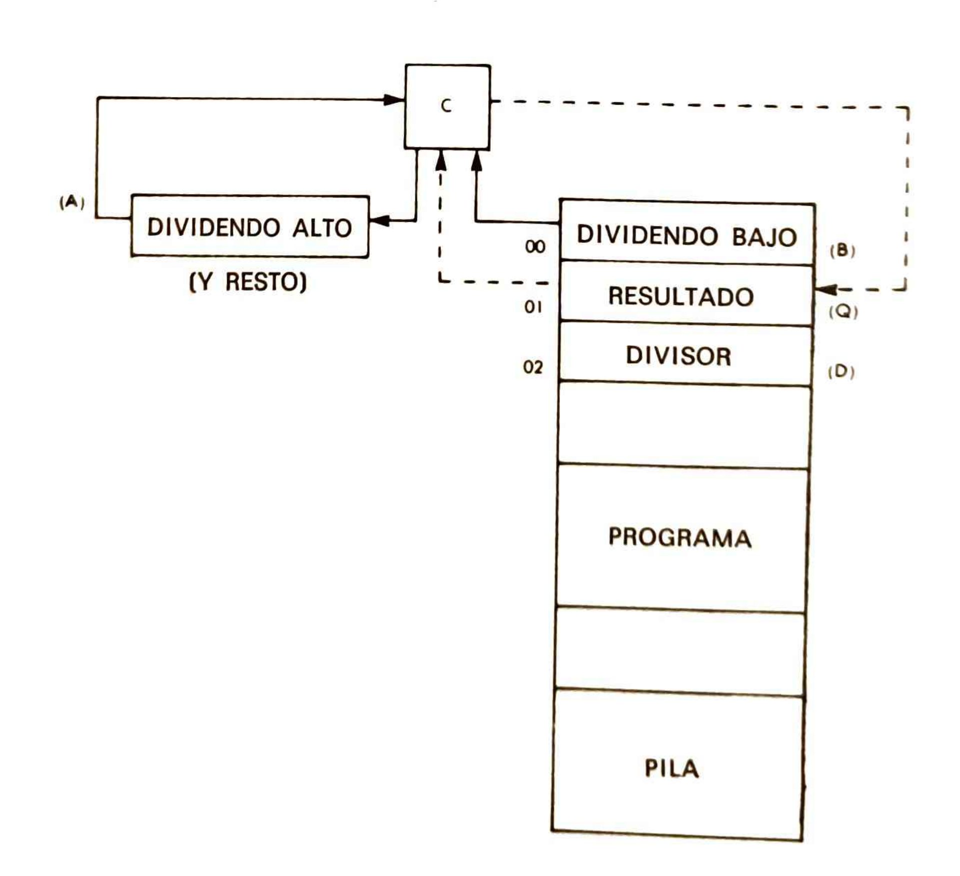


Figura 3-22 Diagrama de flujo de una división 16 por 8 (resultado de 8 bits sin restablecimiento).

Esta instrucción de comparación pone el bit Z (cero) en el registro de indicadores, que se comprobarán después por la siguiente instrucción:

BEQ CERO

La instrucción BEQ significa "bifurcación si es igual". La instrucción de bifurcación determinará si las comprobaciones fueron positivas al examinar el bit Z. Si el bit está posicionado, el programa bifurcará a CERO. Si la comprobación es negativa, entonces se ejecutará la siguiente instrucción secuencial:

CMP #$01

El proceso se repetirá contra la nueva configuración. Si la comprobación es positiva, la siguiente instrucción resultará bifurcada a la posición uno. Si es negativa, se ejecutará la siguiente instrucción secuencial.

Ejercicio 3.16: *Escribir un programa que lea el contenido de la posición de memoria "24" y bifurque a la dirección denominada "STAR", si había un "*" en la posición de memoria 24. La configuración de bits de un "*", en notación de lenguaje ensamblador, se representa por "00101010".*

RESUMEN

Hemos estudiado y utilizado, hasta ahora, la mayoría de las instrucciones importantes del 6502. Hemos transferido valores entre la memoria y los registros. Hemos efectuado operaciones aritméticas y lógicas de tales datos. Las hemos comprobado y, dependiendo de los resultados de estas comprobaciones, hemos ejecutado varias partes de programas. Hemos introducido en el programa de multiplicación una estructura llamada "bucle". Ahora introduciremos una importante estructura de programación: la subrutina.

SUBRUTINAS

Conceptualmente, una subrutina (subprograma) es simplemente un bloque de instrucciones a las que el programador ha dado un nombre. Desde el punto de vista práctico, una subrutina debe comenzar con una instrucción especial llamada la declaración de la subrutina, la cual se identifica como tal por el ensamblador. Se termina también por otra instrucción especial llamada *retorno* (return). Ilustremos, en primer lugar, el empleo de subrutinas en el programa con el fin de mostrar su valor. A continuación, examinaremos cómo se realiza realmente.

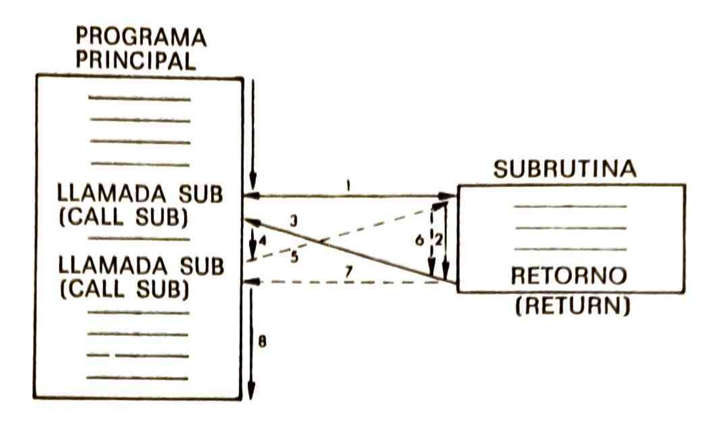


Figura 3-23 Llamadas a subrutinas.

La utilización de una subrutina se ilustra en la figura 3-23. El programa principal aparece a la izquierda de la ilustración. La subrutina se representa simbólicamente a la derecha. Examinemos el mecanismo de la subrutina. Las líneas del programa principal se ejecutan sucesivamente hasta que se encuentra una nueva instrucción, CALL SUB. Esta instrucción especial es la *llamada a subrutina* y resulta de ella una transferencia a la subrutina. Ello significa que la siguiente instrucción que se ejecuta después de la CALL SUB es la primera instrucción en el interior de la subrutina. Esto se ilustra por la flecha 1 en la figura.

A continuación, el subprograma se ejecuta como cualquier otro programa. Supongamos que la subrutina no tiene otras llamadas. La última instrucción de esta subrutina es RETURN (retorno). Esta es una instrucción especial que producirá un retorno al programa principal. La siguiente instrucción que se ejecuta después de RETURN es la siguiente a CALL SUB, que se indica por la flecha 3 en la ilustración. La ejecución del programa continúa después como se indica por la flecha 4.

En el curso del programa principal aparece una segunda CALL SUB. Ocurre una nueva transferencia, mostrada por la flecha 5. Ello significa que el subprograma se ejecuta de nuevo a continuación de la instrucción CALL SUB.

Siempre que se encuentre RETURN en el interior de la subrutina se produce un retorno a la instrucción siguiente a la correspondiente CALL SUB. Esto se representa por la flecha 7. A continuación del retorno al programa principal, la ejecución del programa prosigue normalmente, como se indica por la flecha 8.

La función de las dos instrucciones especiales CALL SUB y RETURN deberá estar ahora claro. ¿Cuál es la importancia de la subrutina?

La importancia principal de la subrutina es que se la puede llamar desde cualquier punto del programa principal y utilizarla repetidamente sin volverla a escribir. Una primera ventaja es que esta consideración ahorra

espacio de memoria y no hay necesidad de reescribir la subrutina cada vez. Una segunda ventaja es que el programador puede concebir una subrutina concreta solamente una vez y después utilizarla repetidamente, por lo que es una simplificación considerable en el diseño de un programa.

Ejercicio 3.17: *¿Cuál es el principal inconveniente de una subrutina?*

El inconveniente de la subrutina deberá aparecer claro al examinar el flujo de la ejecución entre el programa principal y la subrutina. Una subrutina lleva consigo una ejecución más lenta, ya que se deben ejecutar instrucciones adicionales: CALL SUB y RETURN.

Realización del mecanismo de la subrutina

Examinemos cómo se realizan internamente en el procesador las dos instrucciones especiales CALL SUB y RETURN. El efecto de la instrucción CALL SUB es hacer que la siguiente instrucción se busque y cargue en una nueva dirección. Recuerde (si no, lea el capítulo 1 de nuevo) que la dirección de la siguiente instrucción a ejecutar en un ordenador está contenida en el contador de programa (PC). Ello significa que el efecto de CALL SUB es sustituir el nuevo contenido en el registro PC. Su efecto es cargar la dirección de comienzo de la subrutina en el contador de programa. *¿Es esto realmente suficiente?*

Para responder a esta pregunta, consideremos la otra instrucción que se ha realizado: RETURN. La instrucción RETURN debe producir, como su nombre indica, un retorno a la instrucción que sigue a CALL SUB. Esto es posible solamente si la dirección de esta instrucción se ha guardado en alguna parte. Esta dirección será el valor del contador de programa en el momento en que se encontró CALL SUB. Ello es debido a que el contador de programa se incrementa cada vez que se utiliza (lea de nuevo el capítulo 1 si es necesario). Esta es precisamente la dirección que deseamos conservar de modo que podamos posteriormente efectuar RETURN.

El siguiente problema es: ¿En dónde podemos conservar esta dirección de retorno? Esta dirección se debe conservar en una posición razonable, en donde se garantice que no se borrará. Sin embargo, consideremos la situación siguiente, representada en la figura 3-24: en este ejemplo, la subrutina 1 contiene una llamada a SUB 2. Nuestro mecanismo deberá funcionar también en este caso. Naturalmente, podría incluso haber más de dos subrutinas, por ejemplo N llamadas "anidadas". Siempre que se encuentre un nuevo CALL, el mecanismo debe, por tanto, almacenar el contador de programa una vez más. Esto exige que necesitemos al menos 2N posiciones de memoria para este mecanismo. Además, necesitaremos retornar primero desde SUB 2 y

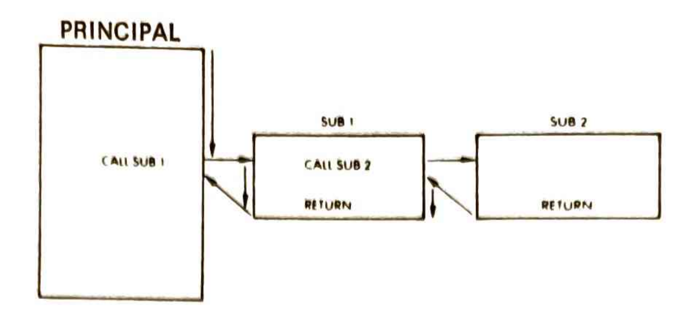


Figura 3-24 Llamadas anidadas.

después desde SUB 1. Dicho de otro modo, necesitamos una estructura que pueda conservar el orden cronológico en el que se habrán salvaguardado los datos.

La estructura tiene un nombre. Lo hemos introducido ya. Es *la pila* (stack). La figura 3-26 muestra el contenido efectivo de la pila durante las sucesivas llamadas a la subrutina. Examinemos, en primer lugar, el programa principal. En la dirección 100 se encuentra la primera llamada: CALL SUB 1. Supongamos que, en este microprocesador, la llamada a subprograma utiliza 3 bytes. La siguiente instrucción secuencial no es, por tanto, "101" sino "103". La instrucción CALL emplea las direcciones "100", "101" y "102". Como la unidad de control del 6502 "sabe" que es una instrucción de 3 bytes, el valor del contador de programas, cuando se haya decodificado completamente la llamada, será "103". El efecto de la llamada será cargar el valor "280" en el contador de programa. "280" es la dirección de comienzo de SUB 1.

El segundo efecto de CALL será introducir en la pila (para preservar)

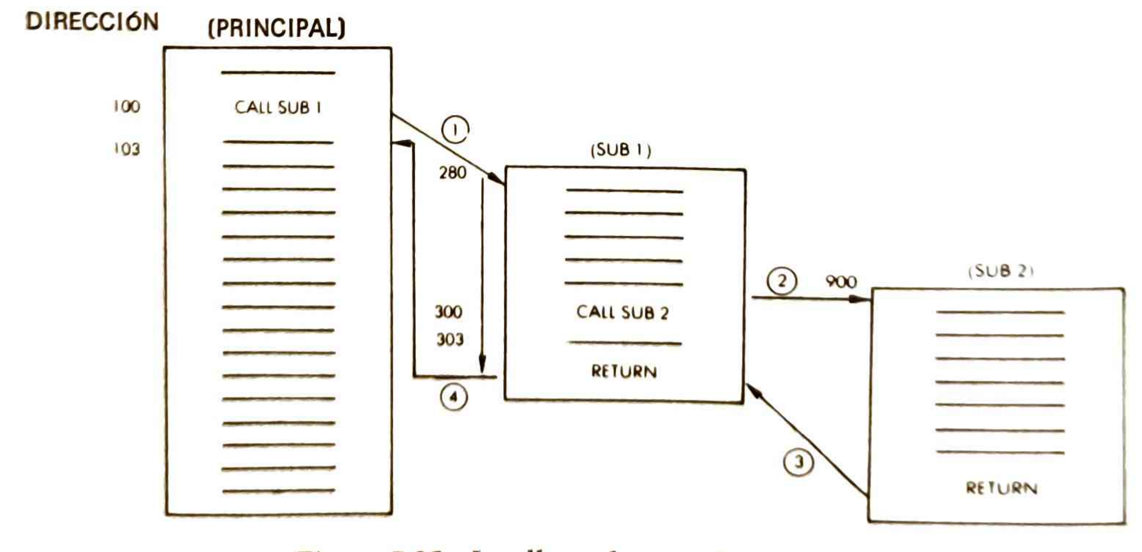


Figura 3-25 Las llamadas a subrutinas.

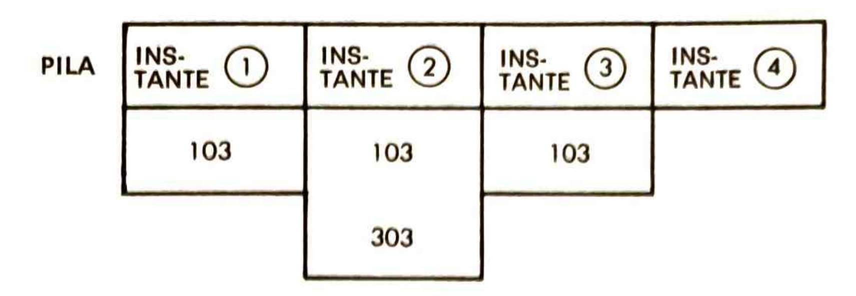


Figura 3-26 La pila en función del tiempo.

el valor "103" del contador de programa. Ello se ilustra en la parte inferior izquierda de la representación que muestra que en el instante 1 el valor "103" se "preserva" en la pila. Desplacémonos a la derecha de la ilustración. En la posición 300, se encuentra una nueva llamada. Al igual que en el caso anterior, el valor "900" se cargará en el contador de programa. Esta es la dirección de comienzo de SUB 2. Simultáneamente, el valor "303" será impulsado a la pila. Esto se ilustra en la parte inferior de la representación en donde la entrada en el instante 2 es "303". La ejecución proseguirá, después, a la derecha de la ilustración en el interior de SUB 2.

Estamos ya en condiciones de poner de manifiesto el efecto de la instrucción RETURN y la operación correcta del mecanismo de nuestra pila. La ejecución continúa en el interior de SUB 2 hasta que se encuentra la instrucción RETURN en el instante 3. El efecto de la instrucción RETURN es simplemente extraer la parte superior de la pila y enviarla al contador de programa. Dicho de otro modo, el contador de programa se restaura a su valor anterior a la entrada de la subrutina. En nuestro ejemplo la cima de la pila es "303". La figura 3-26 muestra que, en el instante 3, el valor "303" se ha desplazado de la pila y se ha puesto de nuevo en el contador de programa. Como resultado de ello, la ejecución de la instrucción prosigue desde la dirección "303". En el instante 4, se encuentra el RETURN de SUB 1. El valor de la cima de la pila es "103". Se extrae y se instala en el contador de programa. Resulta de ello que la ejecución del programa proseguirá desde la posición "103" en el interior del programa principal. Este es realmente el efecto que deseamos. La figura 3-26 muestra que en el instante 4 la pila está vacía de nuevo. El mecanismo funciona adecuadamente.

El mecanismo de llamada a subrutina funciona hasta la máxima dimensión de la pila. Este es el motivo por el que los primitivos microprocesadores, que tenían una pila de 4 u 8 registros, estaban limitados esencialmente a 4 u 8 niveles de llamadas a subrutinas. En teoría, en el 6502, que está limitado a 256 posiciones de memoria de la pila (página 1), se pueden alojar, por tanto, hasta 128 llamadas a subrutinas. Esto solamente es cierto si no hay interrupciones, si la pila no se utiliza para otro fin y si no se necesita

que se almacene ningún registro en la pila. En la práctica, se utilizarán pocos niveles de subrutina.

Obsérvese que en las figuras 3-24 y 3-25, las subrutinas se han mostrado a la derecha del programa principal. Esto es solamente por claridad del diagrama. En realidad, las subrutinas son escritas por el usuario como instrucciones ordinarias del programa. En una hoja de papel de un listado del programa completo, las subrutinas pueden estar al principio, en el medio o al final del texto. Por ello van precedidas de un declaración de subrutina, ya que es necesario identificarlas. Las instrucciones especiales comunican al ensamblador que debe tratar a las que le siguen como una subrutina. Tales *directivos* de ensamblador se presentarán en el capítulo 10.

Subrutinas en el 6502

Hemos descrito el mecanismo de la subrutina y cómo se utiliza la pila para realizarlo. La instrucción de llamada a subrutina del 6502 se llama JSR (jump to subroutine = salto a subrutina). Es, realmente, una instrucción de 3 bytes. Lamentablemente es un salto incondicional: no comprueba los bits. Se deben insertar las bifurcaciones explícitamente antes de un JSR si se necesita realizar una comprobación.

El retorno desde la subrutina es la instrucción RST. Es una instrucción de 1 byte.

Ejercicio 3.18: *¿Por qué es el retorno de una subrutina tan largo como CALL? (Recomendación: si la respuesta no es evidente, vuelva a mirar la realización del mecanismo de la subrutina y analizar las operaciones internas que se deben realizar).*

Ejemplos de subrutinas

La mayoría de los programas que hemos desarrollado, y vamos a desarrollar, se escribirán normalmente como subrutinas. Por ejemplo, es probable que sea utilizado el programa de multiplicación por muchas zonas del programa. Para facilitar el desarrollo del programa y clasificarlo es, por tanto, conveniente definir una subrutina cuyo nombre será, por ejemplo, MULT. Al final de esta subrutina añadiremos simplemente la instrucción RTS.

Ejercicio 3.19: *Si MULT se utiliza como una subrutina, ¿se destruirán los registros o indicadores de estado?*

Recurrencia

La recurrencia o recursión, es una palabra utilizada para indicar que una subrutina se llama a sí misma. Si ha comprendido el mecanismo de realización deberá ahora responder a la siguiente pregunta:

Ejercicio 3.20: *¿Es legal dejar que una subrutina se llame a sí misma? (Dicho de otra forma, ¿funcionará todo incluso si una subrutina se llama a sí misma?). Si no está seguro, dibuje la pila y complétela con las direcciones sucesivas. Verifique físicamente si funciona o no. Esto responderá a la pregunta. Después, observe los registros y la memoria (ver ejercicio 3.19) y determine si existe un problema.*

Parámetros de subrutina

Cuando se llama a una subrutina se suele esperar que la subrutina funcione normalmente con ciertos datos. Por ejemplo, en el caso de la multiplicación, se desea transmitir dos números a la subrutina que realiza la multiplicación. Vimos en el caso de la subrutina de la multiplicación que esta subrutina esperaba encontrar el multiplicando y el multiplicador en las posiciones de memoria dadas. Esto ilustra el primer método del paso de parámetros: a través de la memoria. Se utilizan otras dos técnicas y los parámetros se pueden transmitir de tres formas:

1. A través de los registros.
2. A través de la memoria.
3. A través de la pila.

— *Los registros* pueden ser utilizados para pasar parámetros. Esta es una solución ventajosa siempre que los registros están disponibles, ya que no necesita utilizar una posición de memoria fija. La subrutina queda independiente de la memoria. Si se utiliza una posición fija de memoria, cualquier otro usuario de la subrutina debe tener mucha precaución para utilizar la misma notación y para que la posición de memoria esté efectivamente disponible (consulte el ejercicio 3.20 anterior). Esta es la razón por la que, en muchos casos, se reserva un bloque de posiciones de memoria, simplemente para pasar parámetros entre diferentes subrutinas.

— *El empleo de la memoria* tiene la ventaja de una gran flexibilidad (se pueden transmitir más datos), pero conduce a rendimientos más débiles y también hace depender a la subrutina de una zona de memoria determinada.

— El depósito de parámetros en *la pila* tiene la misma ventaja que la

utilización de registros: es independiente de la memoria. La subrutina "sabe" simplemente que se van a recibir, digamos, dos parámetros que están almacenados en la cima de la pila. Naturalmente, tiene un inconveniente: se rellena la pila con datos y, en consecuencia, reduce el número de niveles posibles de llamadas a subrutinas.

La elección es competencia del programador. En el caso general, se desea permanecer lo más independiente posible de posiciones de memoria determinadas.

Si los registros no están disponibles, la mejor solución que queda suele ser la pila. Sin embargo, si se debe pasar una gran cantidad de información a una subrutina, entonces tendrá que residir esta información en la memoria. Una manera elegante de resolver el problema del paso de un bloque de datos es transmitir simplemente un puntero hacia la información. Un *puntero* es la dirección al principio del bloque. Se puede transmitir un puntero en un registro (ello limita el puntero en el caso del 6502 a 8 bits) o, si no, en la pila (se pueden utilizar dos posiciones de pila para almacenar una dirección de 16 bits).

Por fin, si ninguna de las dos soluciones es aplicable, entonces se puede establecer el convenio en la subrutina de que los datos estarán en una posición de memoria fija (el "buzón").

Ejercicio 3.21: *¿Cuál de los tres métodos anteriores es el mejor para la recurrencia?*

Biblioteca de subrutinas

Hay una ventaja importante en estructurar las partes de un programa en subrutinas identificables: pueden ser depuradas independientemente y pueden tener un nombre nemónico. Si se utilizan en otras zonas del programa, se hacen compartibles y entonces se puede construir una biblioteca de subrutinas útiles. Sin embargo, no hay ninguna panacea general en programación de ordenadores. El empleo sistemático de subrutinas para cualquier juego de instrucciones que puede ser agrupado por función puede conducir a un mal rendimiento de ejecución. El programador tendrá que ponderar equilibradamente las ventajas e inconvenientes.

RESUMEN

Este capítulo ha presentado el modo en que las instrucciones manipulan la información en el interior del 6502. Algoritmos cada vez más complejos

se han introducido y convertido en programas. Se han utilizado los principales tipos de instrucciones.

Se han definido importantes estructuras tales como lazos, pilas y subrutinas.

Ahora, deberá haber adquirido una comprensión básica de programación y de las técnicas principales utilizadas en aplicaciones estándar. Estudiemos las instrucciones disponibles.

4 El juego de instrucciones del 6502

1.ª PARTE. DESCRIPCIÓN GENERAL

INTRODUCCIÓN

En este capítulo se analizará, en primer lugar, las diferentes clases de instrucciones que deberán estar disponibles en un ordenador de aplicación general. Después se analizarán cada una de las instrucciones disponibles en el 6502 y se explicarán en detalle sus aplicaciones y la manera en que afectan a los indicadores de estado y los diversos modos de direccionamiento que se pueden utilizar en conjunción con ellas. En el capítulo 5 se presentará una descripción detallada de las técnicas de direccionamiento.

CLASES DE INSTRUCCIONES

Las instrucciones se pueden clasificar de numerosas formas y no hay clasificación estándar. Distinguiremos aquí cinco categorías principales de instrucciones:

1. Transferencia de datos.
2. Proceso de datos.
3. Comprobación y bifurcaciones.
4. Entrada/salida.
5. Control.

Examinemos cada una de estas clases de instrucciones.

Transferencia de datos

Las instrucciones de transferencia de datos transmitirán un dato de 8 bits entre dos registros, o entre un registro y memoria, o entre un registro y un dispositivo de entrada/salida. Pueden existir instrucciones de transferencia especializadas que desempeñan una función especial, por ejemplo, una operación de introducción y extracción para una manipulación eficaz de la pila. Desplazarán una palabra de datos entre la cima de la pila y el acumulador en una sola instrucción, mientras se actualiza automáticamente el registro de puntero de pila ("stack-pointer").

Proceso de datos

Las instrucciones de proceso de datos se dividen en cuatro categorías generales:

— operaciones aritméticas (tales como más/menos)
— operaciones lógicas (tales como AND, OR, OR exclusiva)
— operaciones de desplazamiento y desalineamiento (tales como desplazamiento, rotación y sustitución)
— incremento y decremento.

Se debe observar que para un proceso de datos eficaz, sería deseable tener instrucciones aritméticas potentes, tales como multiplicación y división. Lamentablemente no están disponibles en la mayoría de los microprocesado-

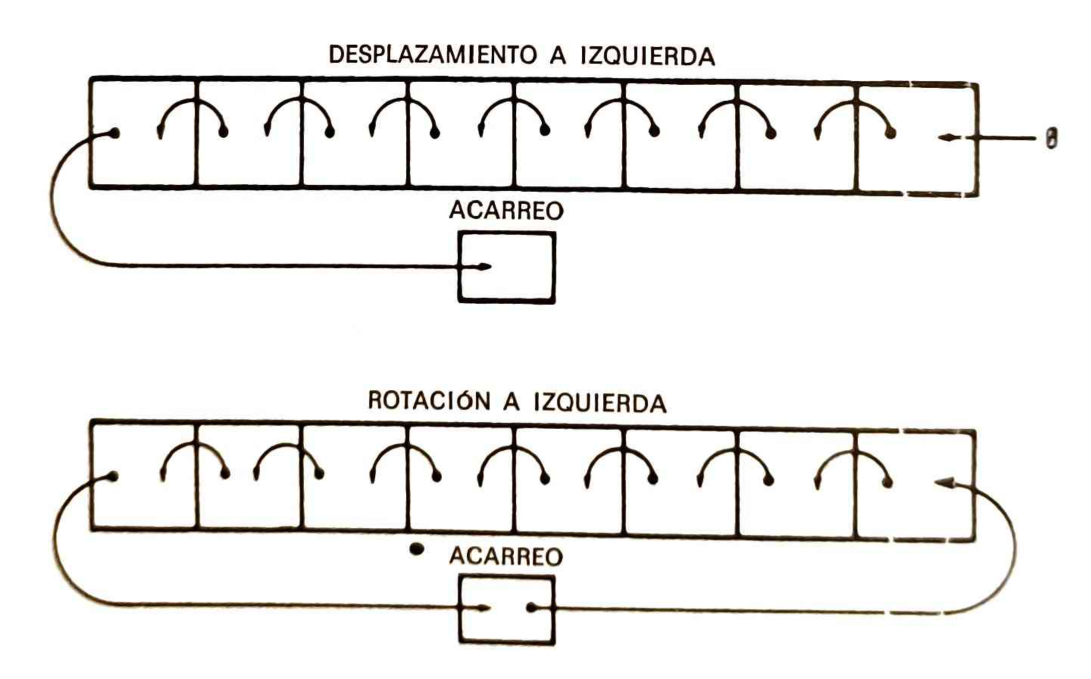


Figura 4-1 Desplazamiento y rotación.

res. También sería deseable tener instrucciones de desplazamiento y desalineamiento (skew), tales como desplazar *n* bits, o un cambio de "nibbles", en donde se intercambian la mitad derecha e izquierda del byte. Ellas no están disponibles en la mayoría de los microprocesadores.

Antes de examinar las instrucciones propias del 6502, recordemos la diferencia entre un *desplazamiento* y una *rotación*. El desplazamiento trasladará el contenido de un registro o una posición de memoria, un bit a la derecha o a la izquierda. El bit que sale del registro irá al bit de acarreo. El bit que entra por el otro lado será un "0".

En el caso de una rotación, el bit que sale va siempre al bit de acarreo. Sin embargo, el bit que entra es el valor antiguo que tenía en el bit de acarreo. Esto corresponde a una rotación de 9 bits. Con frecuencia sería deseable tener una rotación verdadera de 8 bits, en donde el bit que entra por un lado sea el que sale por el otro lado. No suelen disponer de ella los microprocesadores. Finalmente, cuando se desplaza una palabra a la derecha, es conveniente tener otro modo más de desplazamiento llamado *extensión del signo* o un "desplazamiento aritmético a la derecha". Cuando se efectúan operaciones con números en complemento a dos, sobre todo cuando se realizan rutinas de aritmética de coma (punto) flotante, es a menudo necesario desplazar un número negativo a la derecha. Cuando se desplaza a la derecha un número en complemento a dos, el bit que debe entrar por el lado izquierdo debe ser un 1 (el bit de signo se debe repetir cuantas veces sea preciso por desplazamientos sucesivos. Lamentablemente, este tipo de desplazamiento no existe en el 6502. Existe en otros microprocesadores.

Comprobación y bifurcaciones

Las instrucciones de comprobación probarán todos los bits del registro de estado para ver si son "0", "1" o combinación de ellos. Es deseable, en consecuencia, tener tantos indicadores como sea posible en este registro. Además, es conveniente poder comprobar combinaciones de tales bits con una sola instrucción. Finalmente, es deseable poder comprobar cualquier posición de bit en cualquier registro y comprobar el valor de un registro comparado con el valor de cualquier otro registro (mayor, menor o igual). Las instrucciones de comprobación del microprocesador se suelen limitar a comprobar bits aislados del registro de indicadores.

Las instrucciones de salto que pueden estar disponibles se dividen en tres categorías:

— los saltos propiamente dichos, que especifican una dirección completa de 16 bits.

— las bifurcaciones, que se limitan, con frecuencia, a un campo de desplazamiento de 8 bits.
— la llamada (CALL) que se utiliza para las subrutinas.

Es conveniente tener bifurcaciones de dos o incluso tres vías, dependiendo, por ejemplo, de si el resultado de una comparación es "mayor", "menor" o "igual". Es conveniente también tener operaciones de salto, en las que se saltará hacia adelante, o hacia atrás, en unas pocas instrucciones. Finalmente, en la mayoría de los bucles, hay generalmente una operación de decremento o incremento al final, seguida por una comprobación y una bifurcación. La disponibilidad de una sola instrucción que reagrupe incremento/decremento más comprobación y bifurcación es, en consecuencia, una ventaja significativa para la implantación eficaz de los bucles. No está disponible en la mayoría de los microprocesadores. Solamente están disponibles bifurcaciones simples combinadas con comprobaciones simples. Ello complica, naturalmente, la programación y reduce la eficacia.

Entrada/Salida

Las instrucciones de entrada/salida son instrucciones especializadas en el tratamiento de dispositivos de entrada/salida. En la práctica, casi todos los microprocesadores utilizan *memoria correlacionada (mapped) de E/S.* Ello significa que los dispositivos de entrada/salida están conectados al bus de direcciones exactamente como las pastillas de memoria y direccionados como tales. Aparecen ante el programador como posiciones de memoria. Todas las operaciones de tipo memoria se pueden aplicar entonces a los dispositivos deseados. Esto tiene la ventaja de permitir aplicar una amplia gama de instrucciones. El inconveniente es que las operaciones tipo memoria suelen requerir 3 bytes y son, por tanto, lentas. Para un tratamiento eficaz de entrada/salida en tal entorno, es deseable disponer de un mecanismo de direccionamiento corto, de modo que los dispositivos de E/S, cuya velocidad de tratamiento es crucial, pueden residir en la página 0. Sin embargo, si el direccionamiento de página 0 está disponible, se suele utilizar para memoria RAM y, en consecuencia, impide su empleo efectivo para los dispositivos de entrada/salida.

Instrucciones de control

Las instrucciones de control proporcionan señales de sincronización y pueden suspender o interrumpir un programa. Pueden funcionar también como una ruptura o una interrupción simulada. (Las interrupciones se describirán en el capítulo 6 de técnicas de entradas/salidas.)

INSTRUCCIONES DISPONIBLES EN EL 6502

Instrucciones de transferencia de datos

El 6502 tiene un juego completo de instrucciones de transferencia de datos, excepto para la carga del puntero de pila, cuya flexibilidad está limitada. El contenido del acumulador se puede cambiar con una posición de memoria con las instrucciones LDA (carga) y STA (almacenar). Lo mismo se aplica a los registros X e Y. Estas son, respectivamente, las instrucciones LDX, LDY y STX, STY. No hay carga directa de S. Hay, naturalmente, transferencias entre registros: las instrucciones son TAX (transferencia A a X), TAY, TSX, TXA, TXS, TYA. Hay una ligera asimetría, ya que el contenido de la pila se puede cambiar con X, pero no con Y.

No hay operación memoria a memoria de 2 direcciones, tal como "sumar los contenidos de LOC1 y LOC2".

Operaciones con la pila

Dos operaciones de "introducir" (push) y "extraer" (pop) están disponibles. Ellas transfieren el registro A o el registro de estado (P) a la cima de la pila en la memoria, mientras se actualiza el puntero de pila S. Ellas son PHA y PHP. Las instrucciones inversas son PLA y PLP (extraer A y extraer P) que transfiere la cima de la pila respectivamente hacia A o hacia P.

Proceso de datos

Aritmética

El complemento habitual (restringido) de funciones aritméticas, lógicas y desplazamiento está disponible. Las operaciones aritméticas son: ADC, SBC. ADC es una suma con acarreo y no hay suma sin acarreo. Es un inconveniente de poca importancia pues es preciso ejecutar una instrucción CLC antes de cualquier suma. La resta se efectúa por SBC.

Existe un modo decimal especial que permite la suma y resta directas de números expresados en BCD. En muchos otros microprocesadores, solamente una de estas instrucciones BCD está disponible y necesita un código de instrucción distinto. La presencia del indicador de modo decimal multiplica por dos el número de operaciones aritméticas efectivamente disponibles.

Incremento/Decremento

Las operaciones de incremento y decremento están disponibles en la

memoria y en los registros índice X e Y, pero no en el acumulador. Son respectivamente: INC y DEC las que actúan en la memoria; INX, INY y DEX, DEY las que operan en los registros índice X e Y.

Operaciones lógicas

Las operaciones lógicas son las clásicas: AND, ORA, EOR. La función de cada una de estas instrucciones se describirá a continuación.

AND

Cada operación lógica se caracteriza por una tabla de verdad, que proporciona el valor lógico del resultado en función de las entradas. La tabla de verdad de AND se muestra a continuación:

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

La operación AND (Y) se caracteriza por el hecho de que la salida es "1" solamente si ambas entradas son "1". Dicho de otro modo, si una de las entradas es "0" se garantiza que el resultado es "0". Esta característica se utiliza para poner a cero un bit en una palabra. A ello se le llama "enmascaramiento".

Una de las utilizaciones importantes de la instrucción AND es borrar, o enmascarar, selectivamente uno o varios bits en una palabra. Supongamos, por ejemplo, que deseamos poner a cero los cuatro bits más a la derecha de una palabra. Ello será realizado por el programa siguiente:

```
LDA     PALABRA          PALABRA CONTIENE "10101010"
AND     #%11110000       LA MÁSCARA ES "11110000"
```

Supongamos que PALABRA es igual a "10101010". El resultado de este programa es dejar en el acumulador el valor "1010 0000". "%" se utiliza para representar un número binario.

Ejercicio 4.1: *Escribir un programa de tres líneas que ponga a cero los bits 1 y 6 de PALABRA.*

Ejercicio 4.2: *¿Qué produce una máscara igual a "11111111"?*

ORA

Esta instrucción es la operación OR (O) inclusiva. Se caracteriza por la siguiente tabla de verdad:

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

La operación OR lógica se caracteriza por el hecho de que si uno cualquiera de los operandos es "1", el resultado es poner a "1" cualquier bit de una palabra.

```
LDA  #PALABRA
ORA  #%00001111
```

Supongamos que PALABRA contenía "10101010". El valor final del acumulador será "10101111".

Ejercicio 4.3: *¿Qué sucederá si hubiéramos utilizado la instrucción ORA #%10101111?*

Ejercicio 4.4: *¿Cuál es el efecto de una operación lógica OR con "FF" hexadecimal?*

EOR

EOR efectúa la función "OR exclusiva". La OR exclusiva difiere de la OR inclusiva, que acabamos de describir en un aspecto: el resultado es "1" solamente si uno, y sólo uno, de los operandos es igual a "1". Si ambos operandos son iguales a "1", la OR normal dará un resultado "1". La OR exclusiva da un resultado "0". La tabla de verdad es:

0 EOR 0 = 0
0 EOR 1 = 1
1 EOR 0 = 1
1 EOR 1 = 0

La OR exclusiva se utiliza para comparaciones. Si cualquier bit es diferente, la OR exclusiva de dos palabras será distinta de cero. Además, en el caso del 6502, la OR exclusiva se utiliza para *complementar* una palabra, ya que no hay instrucción específica para ello. Esto se hace realizando la EOR de una palabra con todos los dígitos "1". El programa es el siguiente:

```
LDA     #PALABRA
EOR     #%11111111
```

Supongamos que PALABRA contenía "10101010". El valor final del acumulador será "01010101". Podemos verificar que este es el complemento del valor original.

Ejercicio 4.5: *¿Cuál es el efecto de EOR #$00?*

Operaciones de desplazamiento

El 6502 estándar está provisto de un desplazamiento a la izquierda, llamado ASL (arithmetic shift left = desplazamiento aritmético a la izquierda), y un desplazamiento a la derecha, denominado LSR (logical shift right = desplazamiento lógico a la derecha). Se describirán a continuación.

Sin embargo, el 6502 solamente tiene una instrucción de rotación a la izquierda (ROL).

Advertencia: Las versiones antiguas del 6502 tenían una instrucción de rotación adicional. Compruebe la fecha de fabricación para verificar este hecho (ROR = rotación a la derecha).

Comparaciones

Los registros X, Y, A se pueden comparar con la memoria mediante las instrucciones CPX, CPY y CMP.

Comprobación y bifurcación

Ya que la comprobación se realiza casi exclusivamente en el registro de indicadores, examinemos ahora los indicadores disponibles en el 6502. El contenido del registro de indicadores se muestra, a continuación, en la figura 4-2.

Examinemos las funciones de los indicadores de izquierda a derecha.

Signo

El indicador N es idéntico al bit 7 del acumulador, en la mayoría de los casos. Resulta de ello que el bit 7 del acumulador es el único bit que se puede comprobar cómodamente con una sola instrucción. Para comprobar cualquier otro bit del acumulador, es necesario desplazar su contenido. En todos los casos en donde se quiera comprobar rápidamente el contenido de una palabra, la posición del bit más favorable será, en consecuencia, el bit 7.

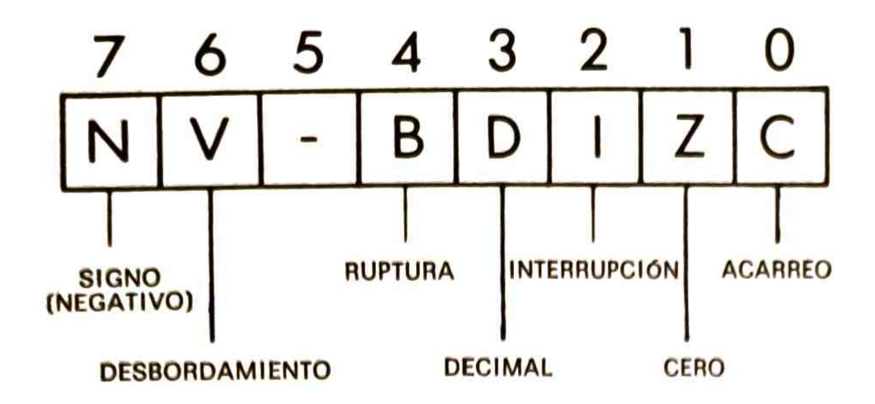


Figura 4-2 El registro de indicadores de estado.

Este es el motivo por el que los bits de estado de las entradas/salidas suelen estar conectados a la posición 7 del bus de datos. Cuando se lee el estado de un dispositivo de E/S, se leerá simplemente en el acumulador el contenido del registro de estado externo y, luego, se comprueba el bit N.

El bit más a la izquierda es el bit de signo o bit negativo. Siempre que N sea 1, indica que el valor del resultado es negativo en representación de complemento a dos. En la práctica, el bit N es idéntico al bit 7 del resultado. Se pone a uno o a cero, por todas las instrucciones de transferencia de datos y proceso de datos.

El segundo bit en el acumulador, que es el más fácil de comprobar, es el bit Z (cero). Sin embargo, requiere un desplazamiento a derecha por 1 en el bit de acarreo de modo que se pueda comprobar.

Las instrucciones que ponen N a uno son: ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, TAX, TAY, TXS, TXA, y TYA.

Desbordamiento

La función del desbordamiento se ha comentado ya en el capítulo 3 en la sección de operaciones aritméticas. Se utiliza para indicar que el resultado de la suma o resta de números en complemento a dos puede ser incorrecto debido a un desbordamiento desde el bit 6 al bit 7, es decir, hacia el bit de signo. Se debe utilizar una rutina de corrección especial cuando este bit se pone a uno. Si no se utiliza la representación en complemento a dos, sino binario natural, el bit de desbordamiento equivale a un acarreo desde el bit 6 al bit 7.

Una utilización especial de este bit se realiza por la instrucción BIT. Un resultado de esta instrucción es posicionar el bit "V" idéntico al bit 6 de los datos que se están comprobando.

El indicador V está condicionado por las instrucciones ADC, BIT, CLV, PLP, RTI y SBC.

Ruptura

Este indicador de ruptura se posiciona automáticamente por el procesador si se produjo una interrupción por el mandato BRK. Hay diferencias entre una ruptura programada y una interrupción de hardware. Ninguna otra instrucción del usuario la modificará.

Decimal

El empleo de este indicador se ha comentado ya en el capítulo 3 en la sección de programas aritméticos. Siempre que D se pone a "1", el procesador trabaja en *modo BCD* y cuando se pone a "0" opera en *modo binario*. Este indicador está condicionado por cuatro instrucciones: CLD, PLP, RTI y SED.

Interrupción

Este bit de máscara de interrupción se puede posicionar explícitamente por el programador con las instrucciones SEI o PLP, o por el microprocesador durante la vuelta al estado inicial (reset) o durante una interrupción.

Su efecto es inhibir cualquier interrupción posterior.

Las instrucciones que condicionan estos bits son: BRK, CLI, PLP, RTI y SEI.

Cero

El indicador Z significa, cuando se posiciona (igual a "1"), que el resultado de una transferencia o una operación es un cero. Se posiciona también por la instrucción de comparación. No hay instrucción concreta que ponga el bit Z a cero o a uno. Sin embargo, se puede obtener fácilmente el mismo resultado. Para posicionar el bit cero, se puede ejecutar, por ejemplo, la siguiente instrucción:

```
LDA #0
```

El bit Z está condicionado por muchas instrucciones: ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, RTI, SBC, TAX, TAY, TXA, TYA.

Acarreo

Se ha visto que el bit de acarreo se utiliza para una aplicación doble. La primera aplicación es indicar un acarreo aritmético o un acarreo negativo durante operaciones aritméticas. Su segunda aplicación es almacenar el bit "que sale fuera" de un registro durante las operaciones de desplazamiento o rotación. Las dos funciones no necesitan ser confundidas necesariamente y no lo están en los grandes ordenadores. Sin embargo, este enfoque ahorra tiempo en el microprocesador, en particular para la realización de una multiplicación o división. El bit de acarreo se puede poner a uno o a cero explícitamente.

Las instrucciones que condicionan el bit de acarreo son: ADC, ASL, CLC, CMP, CPX, CPY, LSR, PLP, ROL, ROR, RTI, SBC, SEC.

Instrucciones de comprobación y bifurcación

En el 6502 no es posible comprobar cada bit del registro indicador de estado a uno o cero. Hay 4 bits que se pueden comprobar y en consecuencia, 8 instrucciones de bifurcación diferentes. Son las siguientes:

— BMI (branch on minus = bifurcación si el resultado es negativo), BPL (branch on plus = bifurcación si el resultado es positivo). Estas dos instrucciones comprueban, naturalmente, el bit N.
— BCC (branch on carry clear = bifurcación si acarreo es cero) y BCS (branch on carry set = bifurcación si acarreo es uno): que comprueban C.
— BEQ (branch when result is null = bifurcación si el resultado es cero) y BNE (branch on result not zero = bifurcación si el resultado no es cero). Comprueban el bit Z.
— BVS (branch when overflow is set = bifurcación si el bit de desbordamiento es 1) y BVC (branch on overflow clear = bifurcación si el bit de desbordamiento es cero). Comprueban el bit V.

Estas instrucciones comprueban y bifurcan dentro de las mismas instrucciones. Todas las bifurcaciones especifican un desplazamiento relativo a la instrucción en curso. Ya que el campo de desplazamiento es 8 bits, éste permite un desplazamiento de -128 a $+127$ (en complemento a dos). El desplazamiento se suma a la dirección de la primera instrucción a continuación de la bifurcación.

Ya que todas las bifurcaciones son de 2 bytes de longitud, resulta de ello un desplazamiento efectivo de $-128 + 2 = -126$ a $+127 + 2 = +129$.

Dos instrucciones más, incondicionales, están disponibles: JMP, y JSR.

JMP es un salto hacia una dirección de 16 bits. JSR es una llamada a subrutina. Salta a una nueva dirección y conserva automáticamente el contador de programa en la pila. Al ser incondicionales, estas dos instrucciones suelen ir precedidas por una instrucción de "comprobación y bifurcación".

Existen dos instrucciones de retorno: RTI, retorno desde interrupción, que se describirá en la sección de interrupciones y RTS, un retorno desde subrutina que extrae una dirección de retorno de la pila (y lo incrementa).

Existen dos instrucciones especiales para comprobación de bits y comparaciones.

La instrucción BIT realiza una operación AND entre la posición de memoria y el acumulador. Un aspecto importante es que *no cambia el contenido del acumulador*. El indicador N se pone al valor del bit 7 de la posición comprobada, mientras que el indicador V se pone al valor del bit 6 de la posición de memoria que se está comprobando. Finalmente, el bit Z indica el resultado de la operación AND. Z se pone a "1" si el resultado es "0". Generalmente se cargará una máscara en el acumulador y después se comprobarán los valores sucesivos de memoria utilizando la instrucción BIT. Si la máscara contiene un solo "1", por ejemplo, éste comprobará si cualquier palabra dada de memoria contiene un "1" en esta posición. En la práctica, ello significa que solamente se utilizará una máscara cuando se comprueban las posiciones de memoria de los bits "0" a "5". Recordará el lector que las posiciones de los bits "6" y "7" son almacenadas automáticamente en los indicadores "V" y "N" respectivamente. No necesitan ser enmascarados.

La instrucción CMP comparará el contenido de la posición de memoria dada por el acumulador, restándola del acumulador. El resultado de la comparación se indicará, en consecuencia, por los bit Z y N. Se puede detectar la igualdad, superioridad o inferioridad. El valor del acumulador no cambia por la comparación. CPX y CPY comparan X e Y respectivamente.

Instrucciones de entrada/salida

No hay instrucciones especiales de entrada/salida en el 6502.

Instrucciones de control

Las instrucciones de control incluyen instrucciones especializadas en poner los indicadores a uno o a cero. Son: CLC, CLD, CLI, CLV que borran los bits C, D, I y V, respectivamente, y SEC, SED, SEI que ponen a uno los bits C, D e I, respectivamente.

La instrucción BRK es la equivalente de una interrupción software y se describirá en el capítulo 6 en la sección de interrupciones.

La instrucción NOP es una instrucción que no tiene efecto y suele utilizarse para ampliar la temporización de un bucle. Por último, dos patillas del 6502 activarán un mecanismo de interrupción y se explicará en el capítulo 6 sobre técnicas de entradas/salidas. Es una posibilidad de control de hardware (patillas IRQ y NMI).

Examinemos cada instrucción en detalle.

Para comprender verdaderamente los diferentes modos de direccionamiento, se insta al lector a leer la parte siguiente, rápidamente la primera vez y detalladamente la segunda vez, tras estudiar el capítulo 5 de técnicas de direccionamiento.

2.ª PARTE. LAS INSTRUCCIONES

ABREVIATURAS

A	Acumulador
M	Dirección de memoria especificada
P	Registro de estado
S	Puntero de pila
X	Registro índice
Y	Registro índice
DATOS	Datos especificados
HEX	Hexadecimal
PC	Contador de programa
PCH	Parte alta del contador de programa
PCL	Parte baja del contador de programa
PILA	Contenido de la cima de la pila
V	OR(O) lógica
Λ	AND(Y) lógica
⊻	OR(O) exclusiva
•	Cambio
←	Toma el valor de (asignación)
( )	Contenido de
(M6)	Bit de posición 6 en la dirección M

ADC

Suma con acarreo

Función: A ← (A) + DATOS + C

Formato:

011bbb01	DIRECCIÓN/DATOS	DIRECCIÓN

Descripción:

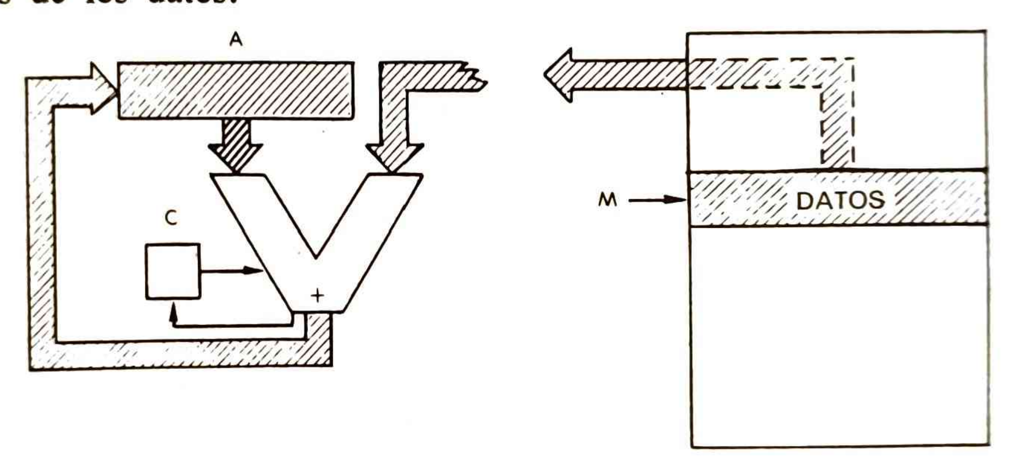
Sumar el contenido de la dirección de memoria o literal al acumulador más el bit de acarreo. El resultado se deja en el acumulador.

Comentarios:

— ADC puede funcionar en modo decimal o bien en modo binario: los indicadores se deben posicionar al valor correcto.

— Para SUMAR sin acarreo, el indicador C se debe poner a cero (CLC).

Caminos de los datos:

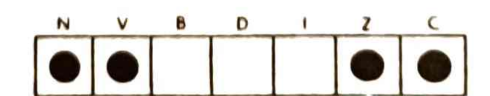


Modos de direccionamiento:

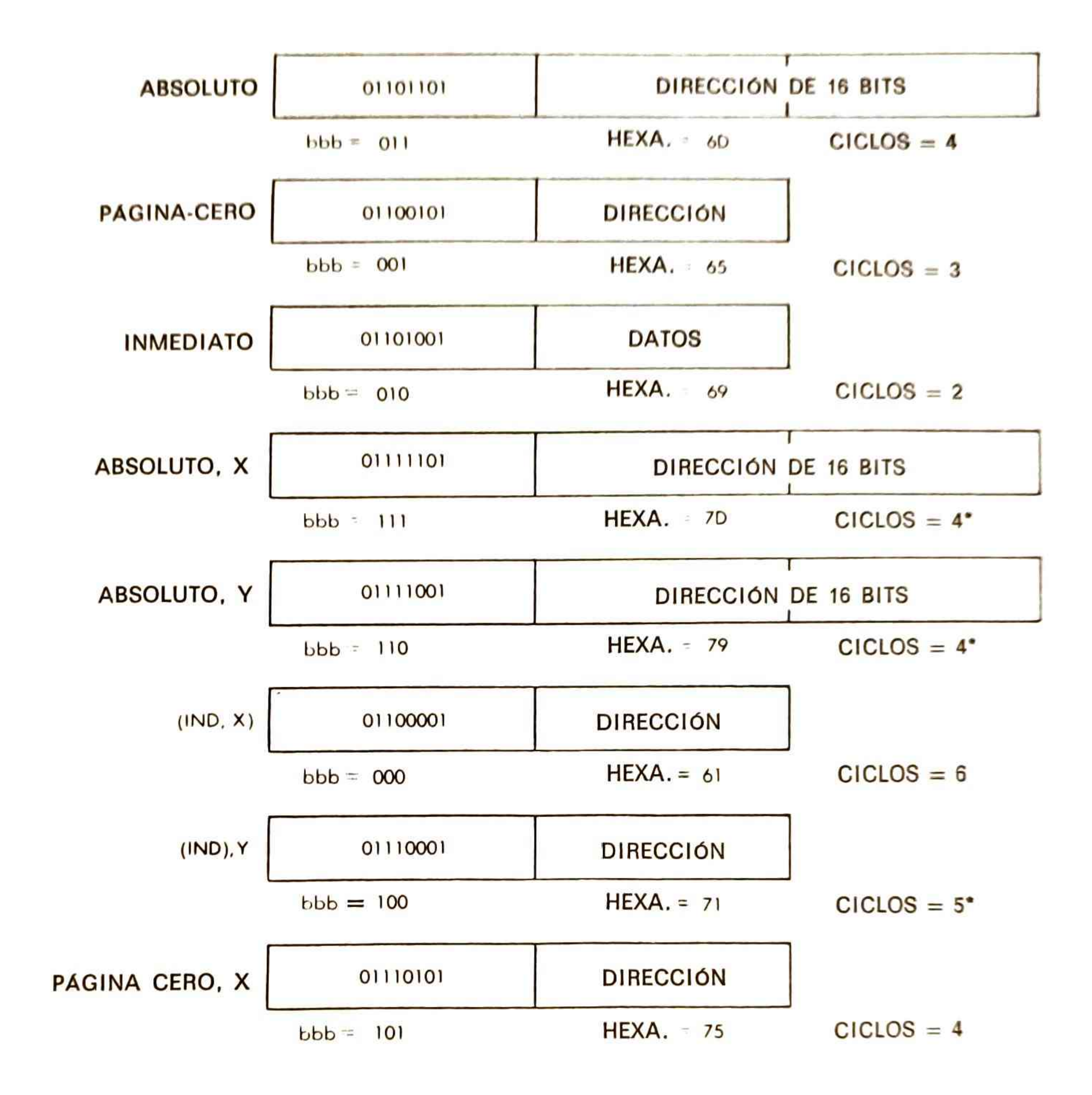
	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS. X	ABS. Y	(IND. X)	(IND), Y	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA.			6D	65	69	7D	79	61	71	75			
BYTES			3	2	2	3	3	2	2	2			
CICLOS			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

*: MAS 1 CICLO SI SE CRUZA EL LIMITE DE LA PAGINA.

Indicadores:



Códigos de instrucción:



*: MÁS 1 CICLO SI SE CRUZA EL LÍMITE DE LA PÁGINA.

AND

AND lógica

Función: A ← (A) Λ DATOS

Formato:

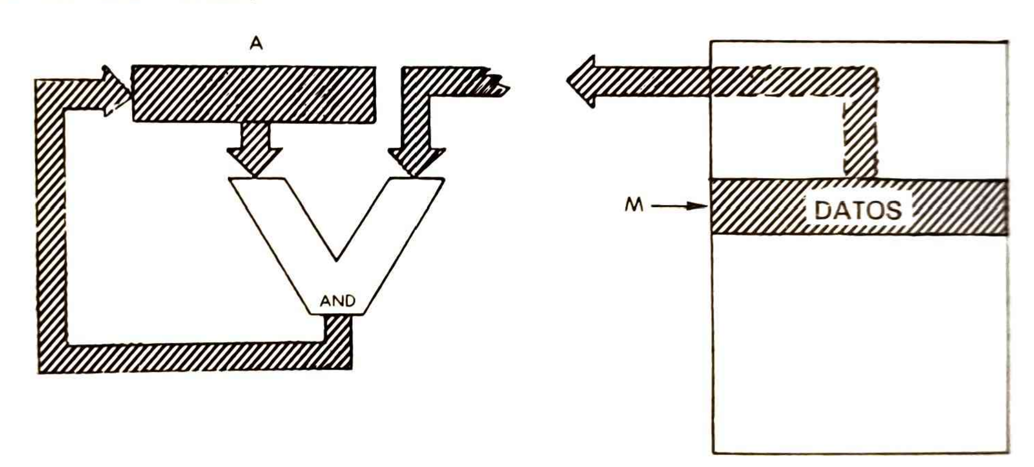
001bbb01	DIRECCIÓN/DATOS	DIRECCIÓN

Descripción:

Realiza la operación lógica AND entre el acumulador y la dirección especificada. El resultado se deja en el acumulador.
La tabla de verdad es:

A\M	0	1
0	0	0
1	0	1

Caminos de los datos:

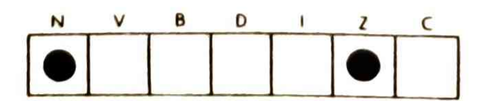


Modos de direccionamiento:

	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS. X	ABS. Y	(IND. X)	(IND), Y	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA.			2D	25	29	3D	39	21	31	35			
BYTES			3	2	2	3	3	2	2	2			
CICLOS			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

*: MAS 1 CICLO SI SE CRUZA EL LÍMITE DE LA PAGINA.

Indicadores:



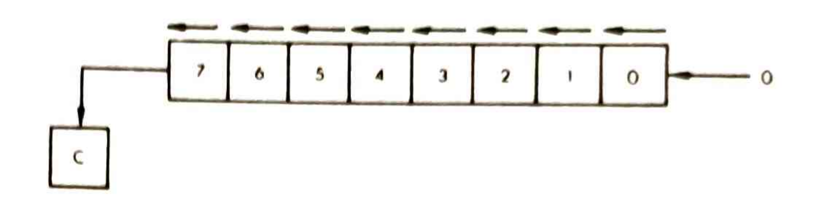
Códigos de instrucción:

Modo	Código	Operando	bbb	HEXA.	CICLOS
ABSOLUTO	00101101	DIRECCIÓN DE 16 BITS	bbb = 011	HEXA. = 2D	CICLOS = 4
PAGINA-CERO	00100101	DIRECCIÓN	bbb = 001	HEXA. = 25	CICLOS = 3
INMEDIATO	00101001	DATOS	bbb = 010	HEXA. = 29	CICLOS = 2
ABSOLUTO, X	00111101	DIRECCIÓN DE 16 BITS	bbb = 111	HEXA. = 3D	CICLOS = 4*
ABSOLUTO, Y	00111001	DIRECCIÓN DE 16 BITS	bbb = 110	HEXA. = 39	CICLOS = 4*
(IND, X)	00100001	DIRECCIÓN	bbb = 000	HEXA. = 21	CICLOS = 6
(IND),Y	0011001	DIRECCIÓN	bbb = 100	HEXA. = 31	CICLOS = 5*
PÁGINA CERO, X	00110101	DIRECCIÓN	bbb = 101	HEXA. = 35	CICLOS = 4

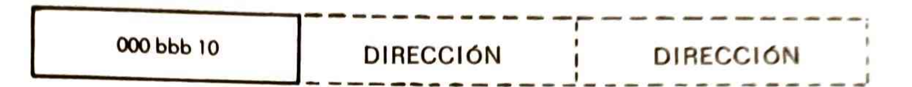
*: MÁS 1 CICLO SI SE CRUZA EL LIMITE DE LA PÁGINA.

ASL   Desplazamiento aritmético a la izquierda

Función:



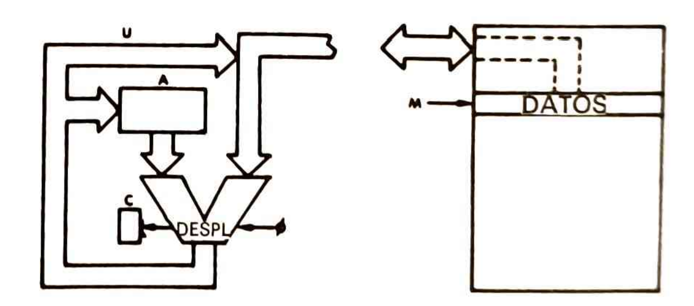
Formato:



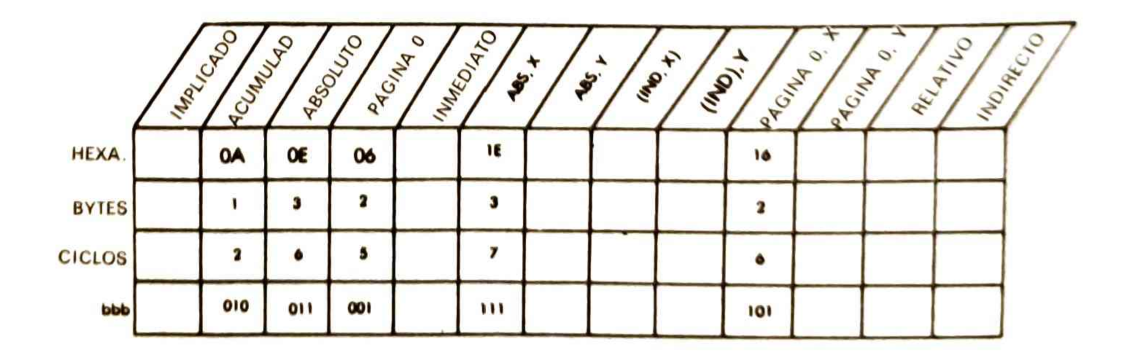
Descripción:

Desplaza el contenido del acumulador, o de la posición de memoria, en una posición de un bit a la izquierda. Un 0 entra por la derecha. El bit 7 va al indicador de acarreo. El resultado se deposita en la fuente, es decir, ya sea el acumulador o la memoria.

Caminos de los datos:

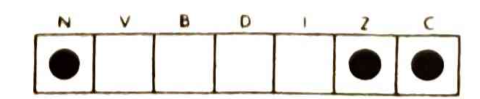


Modos de direccionamiento:

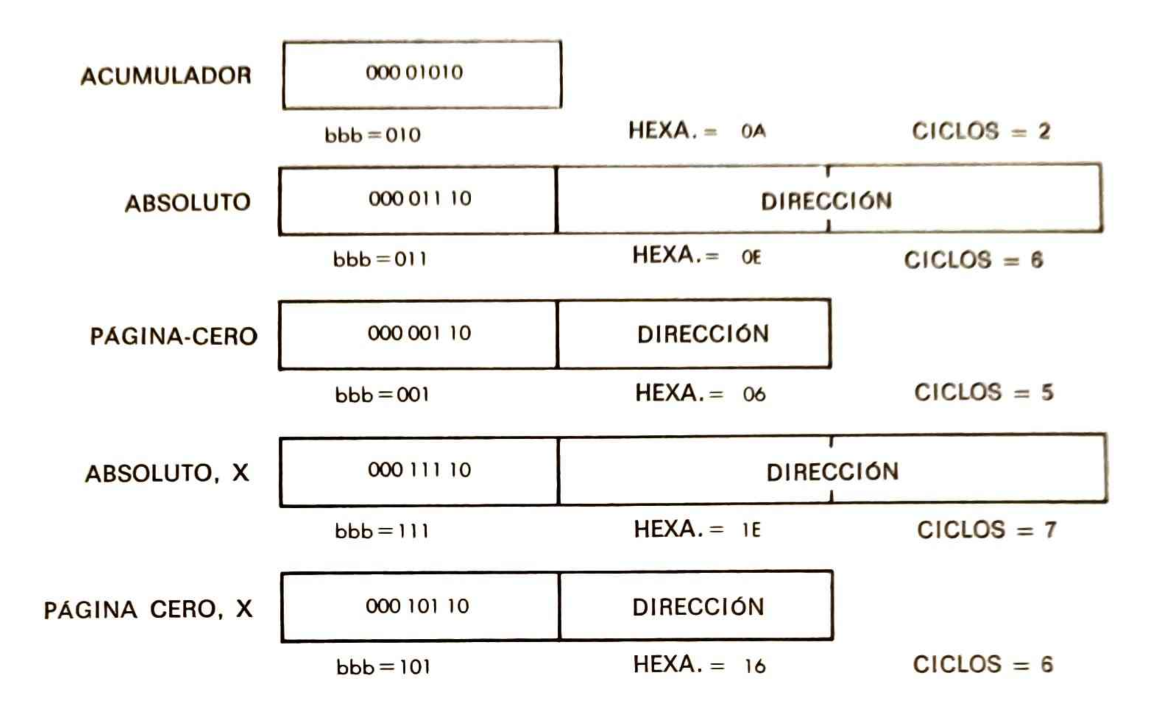


	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS, X	ABS, Y	(IND, X)	(IND), Y	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA.		0A	0E	06		1E				16			
BYTES		1	3	2		3				2			
CICLOS		2	6	5		7				6			
bbb		010	011	001		111				101			

Indicadores:



Códigos de instrucción:



BCC   Bifurcación si el acarreo es cero

Función:

Va a la dirección indicada si C = 0.

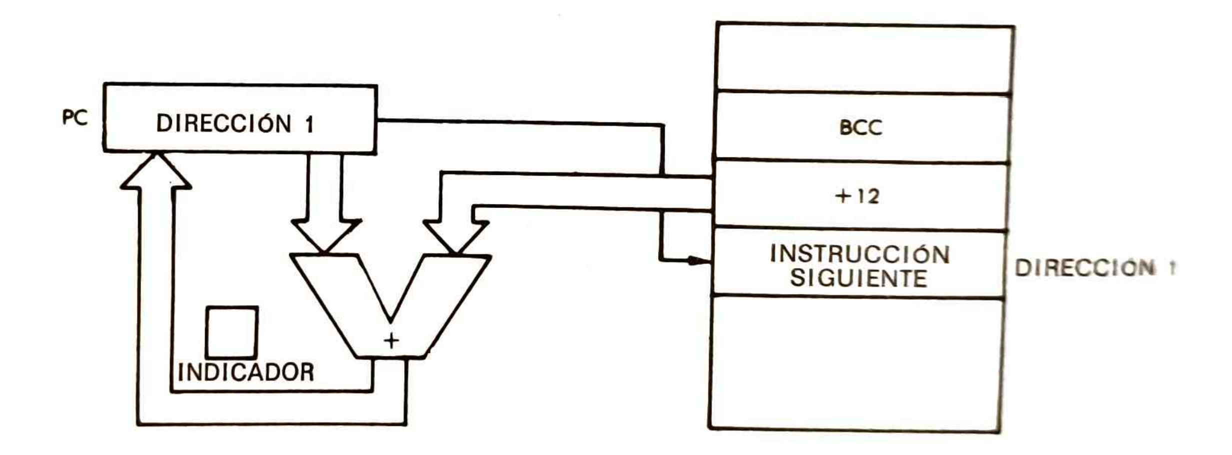
Formato:

1001000	DESPLAZAMIENTO

Descripción:

Comprueba el indicador de acarreo. Si C = 0, bifurca a la dirección actual más el desplazamiento con signo (entre +127 y −128). Si C = 1 no se hace nada. El desplazamiento se suma a la dirección de la primera instrucción a continuación de BCC. Resulta de ello un desplazamiento efectivo de +129 a −126.

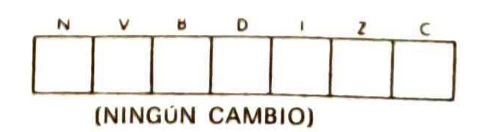
Caminos de los datos:



Modos de direccionamiento:

Solamente relativo:
HEX = 90, bytes = 2, ciclos = 2 + 1 si tiene lugar la bifurcación
+ 2 si se cambia de página

Indicadores:



BCS **Bifurcación si el acarreo es uno**

Función:

Va a la dirección indicada si C = 1.

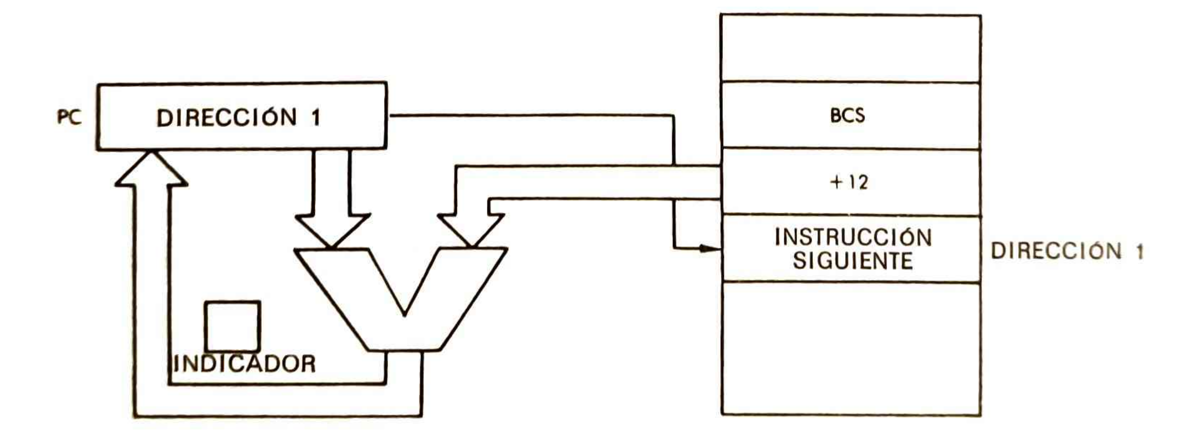
Formato:

10110000	DESPLAZAMIENTO

Descripción:

Comprueba el indicador de acarreo. Si C = 1, bifurca a la dirección actual más el desplazamiento con signo (entre +127 y −128). Si C = 0, no hace nada. El desplazamiento se añade a la dirección de la primera instrucción a continuación de BCS. Resulta de ello un desplazamiento efectivo de +129 a −126.

Caminos de los datos:



Modo de direccionamiento:

Solamente relativo:

HEX = B0, bytes = 2, ciclos = 2 + 1 si tiene lugar la bifurcación
+ 2 si cambia de página

Indicadores:



BEQ Bifurcación si el resultado es igual a cero

Función:

Va a la dirección indicada si Z = 1 (resultado = 0).

Formato:

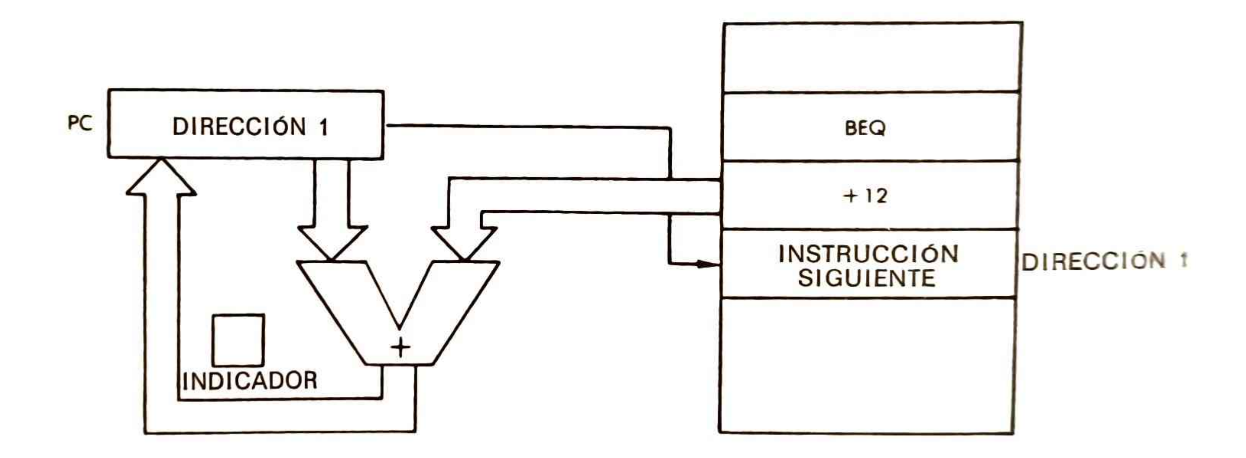
11110000	DESPLAZAMIENTO

Descripción:

Comprueba el indicador Z. Si Z = 1, bifurca a la dirección actual más el desplazamiento con signo (entre +127 y −128). Si Z = 0, no hace nada, o sea, se pasa a la instrucción siguiente.

El desplazamiento se añade a la dirección de la primera instrucción que sigue a BEQ. Este resultado es un desplazamiento efectivo de +129 a −126.

Caminos de los datos:



Modo de direccionamiento:

Solamente relativo:

HEX = F0, bytes = 2, ciclos = 2 + 1 si se produce bifurcación
+ 2 si se cambia de página

Indicadores:



BIT Comparar bits de memoria con acumulador

Función:

$Z \leftarrow \overline{(A) \wedge (M)}$, $N \leftarrow (M^7)$, $V \leftarrow (M^6)$

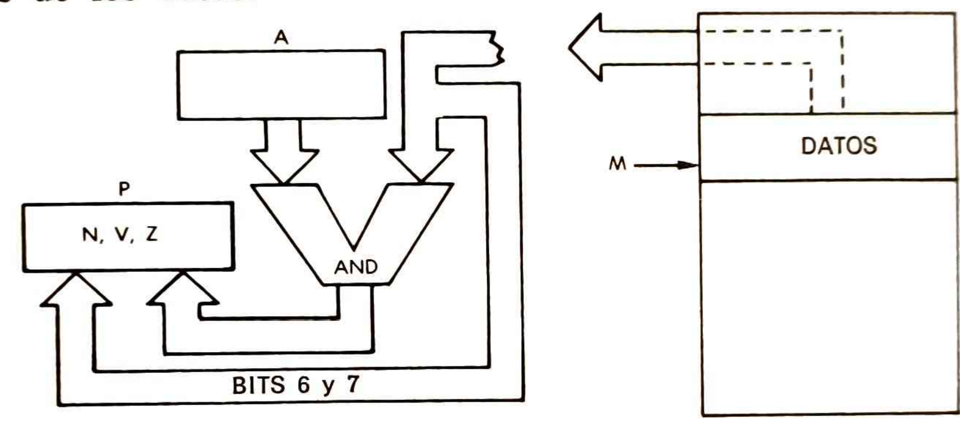
Formato:

0010b100	DIRECCIÓN	DIRECCIÓN

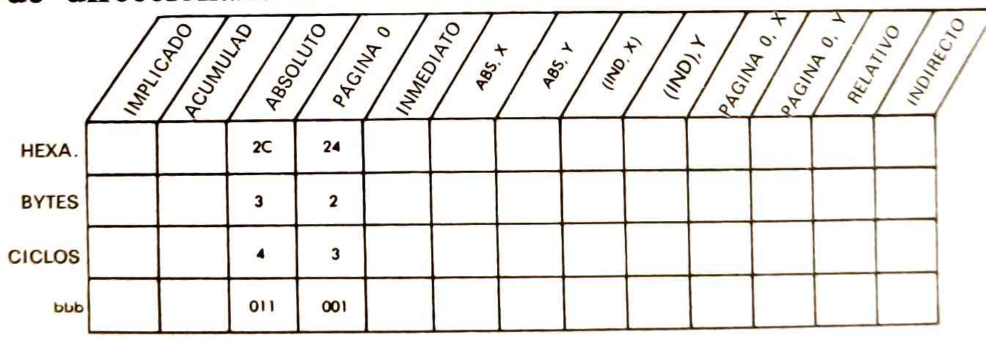
Descripción:

Se efectúa la operación lógica AND de A y M, pero no se almacena. El resultado de la comparación se indica por Z. Z = 1 si no tiene lugar la comparación; 0, en caso contrario. Además, los bits 6 y 7 de la memoria de datos se transfieren a los bits V y N del registro de estado. No se modifica el contenido de A.

Caminos de los datos:

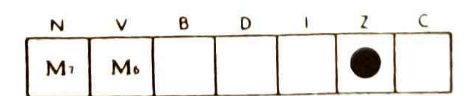


Modos de direccionamiento:



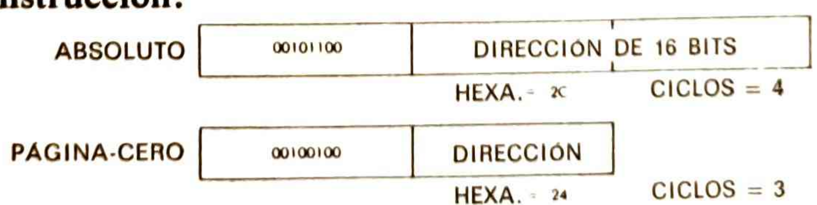
	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS. X	ABS. Y	(IND. X)	(IND), Y	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA.			2C	24									
BYTES			3	2									
CICLOS			4	3									
bbb			011	001									

Indicadores:



N	V	B	D	I	Z	C
M_7	M_6				●	

Códigos de instrucción:



BMI

Bifurcación si el resultado es negativo

Función:

Va a la dirección indicada si N = 1 (resultado < 0).

Formato:

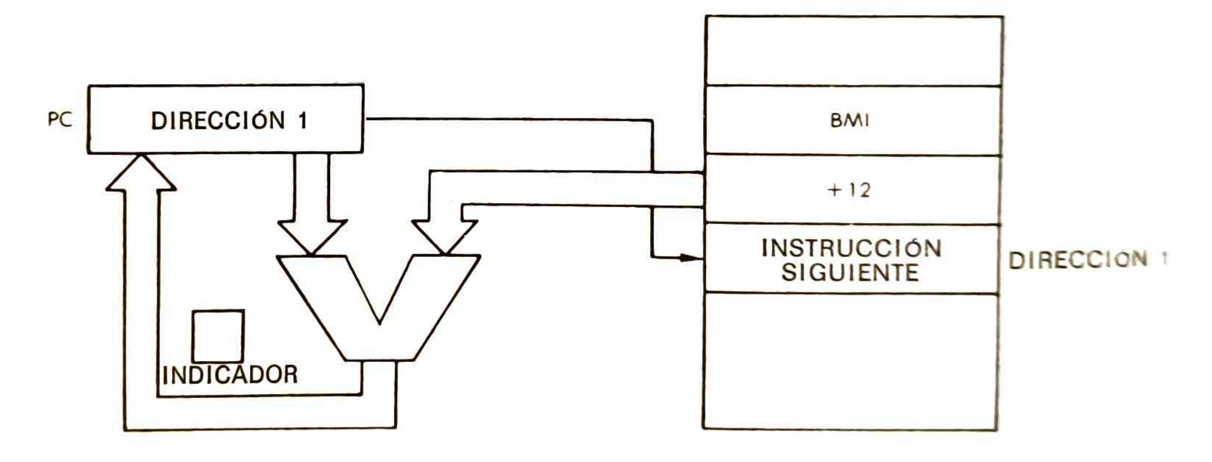
00110000	DESPLAZAMIENTO

Descripción:

Comprueba el indicador N (signo). Si N = 1, bifurca a la dirección actual más el desplazamiento con signo (de +127 a −128). Si N = 0 no tiene acción.

El desplazamiento se añade a la dirección de la primera instrucción que sigue a BMI. Este resultado es un desplazamiento efectivo de +129 a −126.

Caminos de los datos:

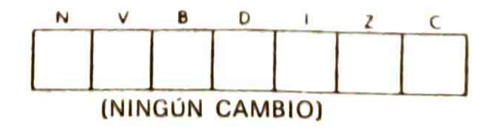


Modo de direccionamiento:

Solamente relativo:

HEX = 30, bytes = 2, ciclos = 2 + 1 si se produce la bifurcación
+ 2 si se cambia de página

Indicadores:



BNE Bifurcación si el resultado no es igual a cero

Función:

Va a la dirección indicada si Z = 0 (resultado ≠ 0).

Formato:

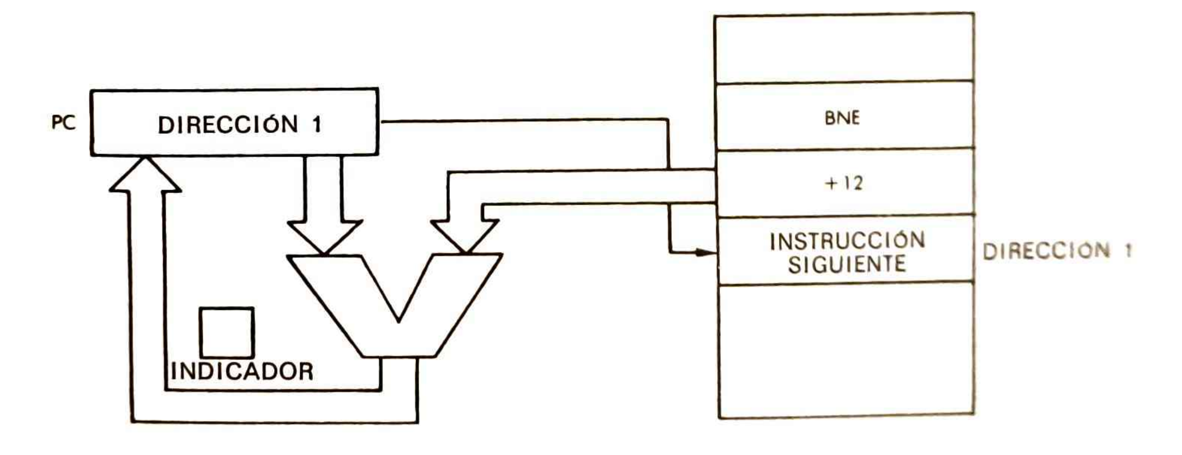
11010000	DESPLAZAMIENTO

Descripción:

Comprueba el resultado (indicador Z). Si el resultado no es igual a 0 (Z = 0), bifurca a la dirección actual más el desplazamiento con signo (entre +127 a −128). Si Z = 1 no tiene acción.

El desplazamiento se añade a la dirección de la primera instrucción que sigue a BNE. Resulta de ello un desplazamiento efectivo de +129 a −126.

Caminos de los datos:



Modo de direccionamiento:

Solamente relativo:

HEX = D0, bytes = 2, ciclos = 2 + 1 si la bifurcación tiene lugar
+ 2 si se cambia de página

Indicadores:



BPL — Bifurcación si el resultado es positivo

Función:

Va a la dirección indicada si N = 0 (resultado ⩾ 0).

Formato:

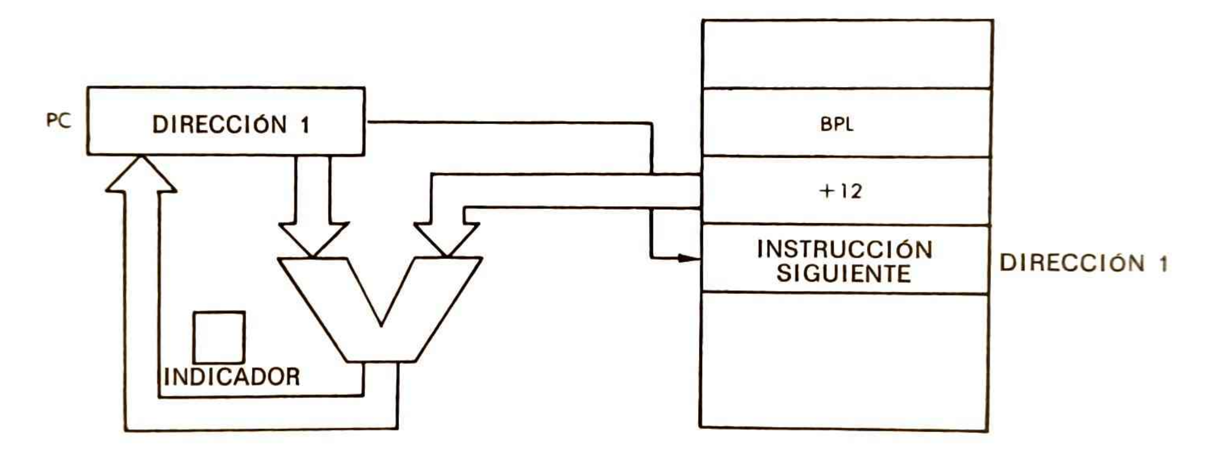
00010000	DESPLAZAMIENTO

Descripción:

Comprueba el indicador N (signo). Si N = 0 (resultado positivo), bifurca a la dirección actual más el desplazamiento con signo (de + 127 a − 128). Si N = 1, no tiene acción.

El desplazamiento se añade a la dirección de la primera instrucción siguiente a BPL. Este resultado es un desplazamiento efectivo de +129 a −126.

Caminos de los datos:

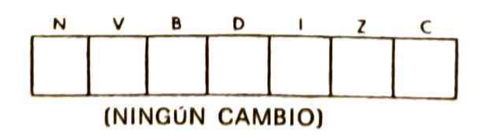


Modo de direccionamiento:

Solamente relativo:

HEX = 10, bytes = 2, ciclos = 2 + 1 si la bifurcación se realiza
+ 2 si se cambia a otra página

Indicadores:



BRK

Ruptura (Break)

Función:

PILA (PC) + 2, PILA (P), PC ← (FFFE, FFFF)

Formato:

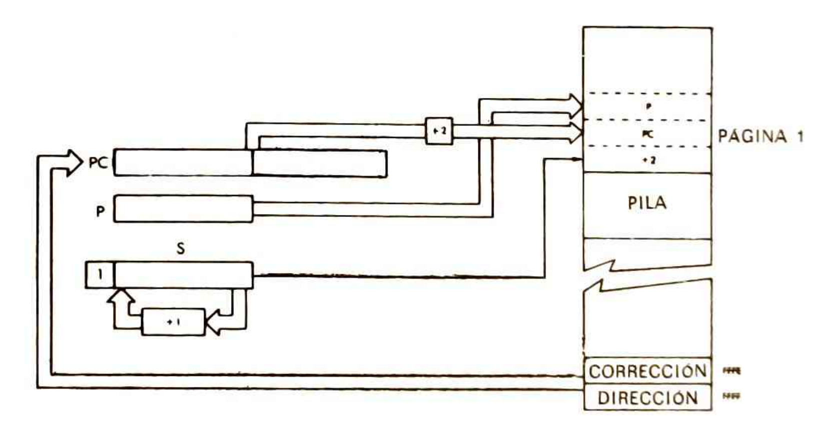
00000000

Descripción:

Funciona como una interrupción: el contador de programa se introduce en la pila y luego en el registro de estado P. Los contenidos de las posiciones de memoria FFFE y FFFF son depositados respectivamente en PCL y PCH. El valor de P almacenado en la pila tiene el indicador B puesto a 1, para diferenciar BRK de IRQ.

Importante: A diferencia de una interrupción, se reserva PC + 2. Esta puede no ser la instrucción subsiguiente y puede ser necesaria una corrección. Ello se debe al empleo normal de BRK para componer programas existentes en donde BRK sustituye a una instrucción de 2 bytes. Cuando se depura un programa, BRK se utiliza generalmente para causar una salida a monitor. Entonces, BRK remplaza a menudo el primer byte de una instrucción.

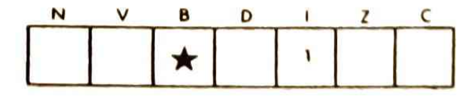
Caminos de los datos:



Modo de direccionamiento:

Solamente implícito:
HEX = 00, byte 1, ciclos = 7

Indicadores:



Nota: B se pone a uno antes de meter P en la pila.

BVC **Bifurcación si el desbordamiento es cero**

Función:

Va a la dirección indicada si V = 0.

Formato:

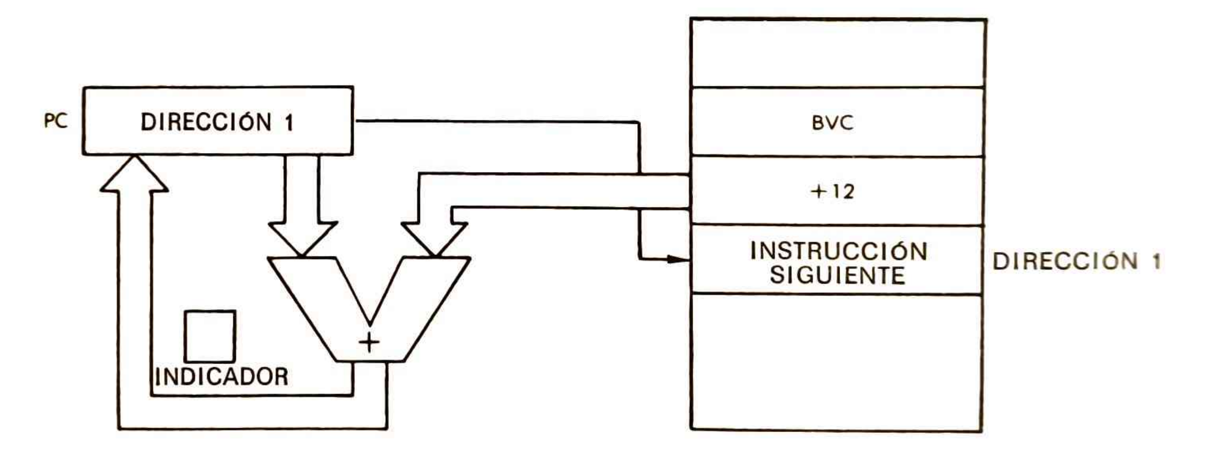
0101000	DESPLAZAMIENTO

Descripción:

Comprueba el indicador de desbordamiento (V). Si no hay desbordamiento (V = 0), bifurca a la dirección actual más el desplazamiento con signo (desde +127 a −128). Si V = 1, no tiene acción.

El desplazamiento se añade a la dirección de la primera instrucción que sigue a BVC. Este resultado es un desplazamiento efectivo de +129 a −126.

Caminos de los datos:

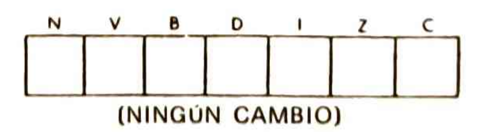


Modo de direccionamiento

Solamente relativo:

HEX = 50, bytes = 2, ciclos = 2 + 1 si se produce la bifurcación
+ 2 si se cambia de página

Indicadores:



BVS — Bifurcación si el desbordamiento es uno

Función:

Va a la dirección especificada si V = 1.

Formato:

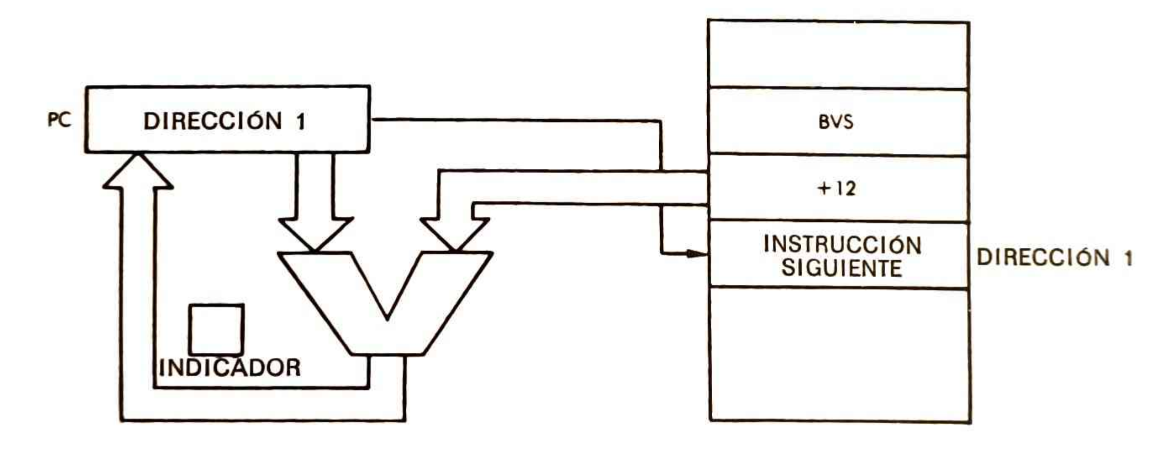
01110000	DESPLAZAMIENTO

Descripción:

Comprueba el indicador de desbordamiento (V). Si se ha producido un desbordamiento (V = 1), se bifurca a la dirección actual más el desplazamiento con signo (desde +127 a −128). Si V = 0, no tiene acción.

El desplazamiento se añade a la dirección de la primera instrucción a continuación de BVS. Este resultado es un desplazamiento efectivo de +129 a −126.

Caminos de los datos:

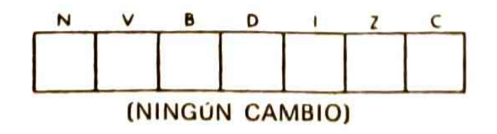


Modo de direccionamiento:

Solamente relativo:

HEX = 70, bytes = 2, ciclos = 2 + 1 si se realiza la bifurcación
+ 2 si se cambia de página

Indicadores:



CLC

Puesta a cero del acarreo

Función:

C ← Ø

Formato:

00011000

Descripción:

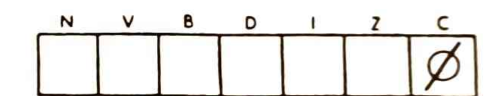
El bit de acarreo se pone a 0. Ello suele ser necesario antes de ADC.

Modo de direccionamiento

Solamente implícito:

HEX = 18, byte = 1, ciclos = 2

Indicadores:



CLD Puesta a cero del indicador de modo decimal

Función:

D ← Ø

Formato:

11011000

Descripción:

El indicador D se pone a 0, lo que establece el modo binario para ADC y SBC.

Modo de direccionamiento:

Solamente implícito:
HEX = D8, byte = 1, ciclos = 2

Indicadores:

N	V	B	D	I	Z	C
			Ø			

CLI Puesta a cero de la máscara de interrupciones

Función:

I ← Ø

Formato:

01011000

Descripción:

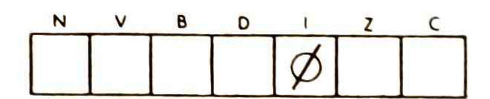
El bit de máscara de interrupciones se pone a 0. Ello habilita las interrupciones. Una rutina de tratamiento de la interrupción debe borrar siempre el bit I o, de no ser así, se pueden perder otras interrupciones.

Modo de direccionamiento:

Solamente implícito:

HEX = 58, byte = 1, ciclos = 2

Indicadores:



CLV **Puesta a cero del indicador de desbordamiento**

Función:

V ← Ø

Formato:

10111000

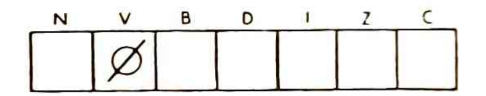
Descripción:

El indicador de desbordamiento es puesto a cero.

Modo de direccionamiento:

Solamente implícito:
HEX = B8, byte = 1, ciclos = 2

Indicadores:



CMP

Comparar con el acumulador

Función:

(A) − DATOS → NZC:

+ (A > DATOS)	=	− (A < DATOS)
- 01	011	- 00

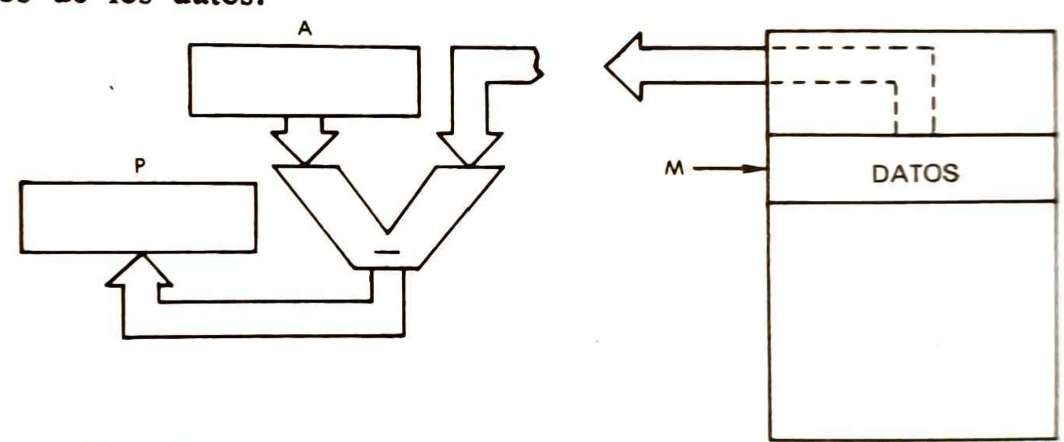
Formato:

110bbb01	DIRECCIÓN/DATOS	DIRECCIÓN

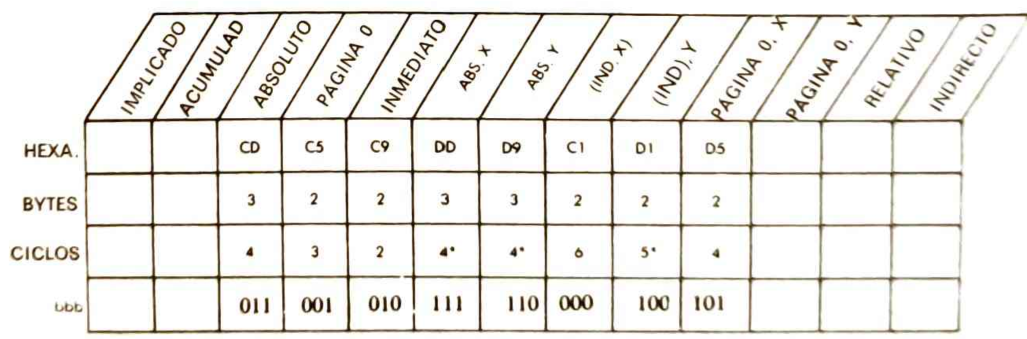
Descripción:

El contenido indicado se resta de A. El resultado no se almacena, sino que los indicadores NZC son posicionados según que el resultado sea positivo, nulo o negativo. El valor del acumulador no se cambia. Z se posiciona por una igualdad y se pone a cero en caso contrario; N se posiciona por el bit de signo (7) y se pone a cero; C se posiciona cuando (A) ⩾ DATOS. CMP suele ir seguida por una bifurcación: BCC detecta A < DATOS, BEQ detecta A = DATOS, BCS detecta A ⩾ DATOS y BEQ seguida por BCS detecta A ⩾ DATOS.

Caminos de los datos:



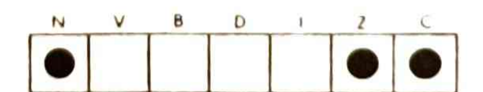
Modos de direccionamiento:



	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS. X	ABS. Y	(IND. X)	(IND), Y	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA.			CD	C5	C9	DD	D9	C1	D1	D5			
BYTES			3	2	2	3	3	2	2	2			
CICLOS			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

*: MAS 1 CICLO SI SE CRUZA EL LIMITE DE LA PAGINA.

Indicadores:



Códigos de instrucción:

Modo	Código	Operando	bbb	HEXA.	CICLOS
ABSOLUTO	11001101	DIRECCIÓN DE 16 BITS	011	CD	4
PAGINA-CERO	11000101	DIRECCIÓN	001	C5	3
INMEDIATO	11001001	DATOS	010	C9	2
ABSOLUTO, X	11011101	DIRECCIÓN DE 16 BITS	111	DD	4*
ABSOLUTO, Y	11011001	DIRECCIÓN DE 16 BITS	110	D9	4*
(IND, X)	11000001	DIRECCIÓN	000	C1	6
(IND),Y	11010001	DIRECCIÓN	100	D1	5*
PÁGINA CERO, X	11010101	DIRECCIÓN	101	D5	4

*: MÁS 1 CICLO SI SE CRUZA EL LÍMITE DE LA PÁGINA.

CPX

Comparar con registro X

Función:

X − DATOS → NZC:

+ (X > DATOS)	=	− (X < DATOS)
−01	011	−00

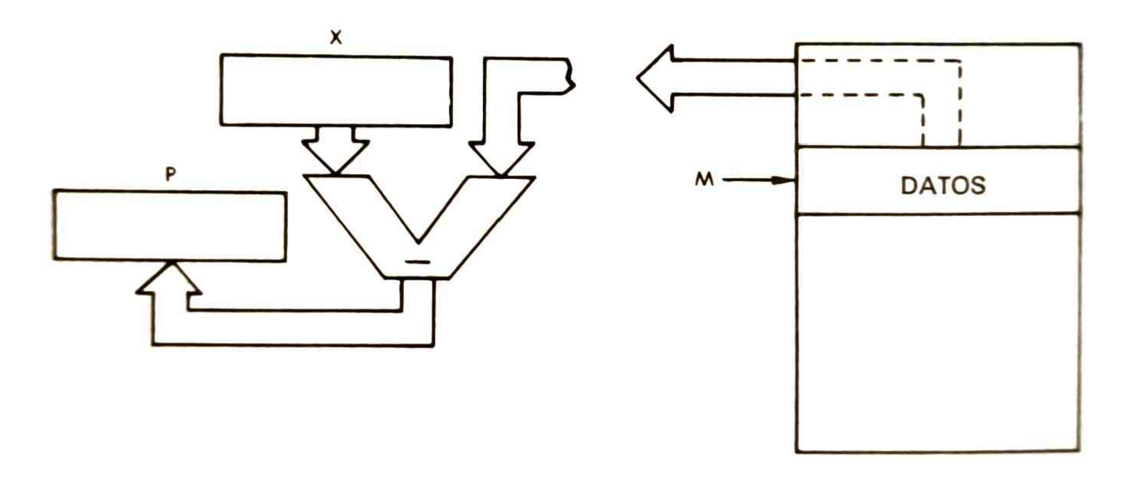
Formato:

111φbbφφ	DIRECCIÓN/DATOS	DIRECCIÓN

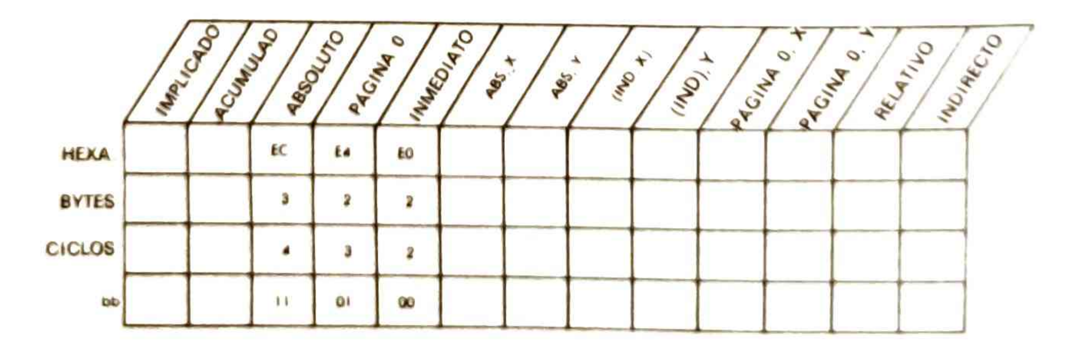
Descripción:

El contenido indicado se resta de X. El resultado no se almacena, sino que los indicadores NZC se posicionan según que el resultado sea positivo, nulo o negativo. El valor del acumulador no cambia. CPX suele ir seguida por una bifurcación: BCC detecta X < DATOS, BEQ detecta X = DATOS y BEQ seguida por BCS detecta X > DATOS. BCS detecta X ⩾ DATOS.

Caminos de los datos:

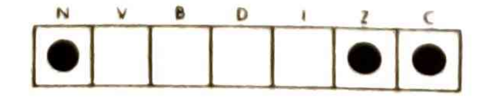


Modos de direccionamiento:



	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS, X	ABS, Y	(IND X)	(IND), Y	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA			EC	E4	E0								
BYTES			3	2	2								
CICLOS			4	3	2								
bb			11	01	00								

Indicadores:



Códigos de instrucción:

ABSOLUTO	11101100	DIRECCIÓN DE 16 BITS
	bb = 11	HEXA. = EC	CICLOS = 4
PÁGINA-CERO	11100100	DIRECCIÓN
	bb = 01	HEXA. = E4	CICLOS = 3
INMEDIATO	11100000	DATOS
	bb = 00	HEXA. = EO	CICLOS = 2

CPY **Comparar con registro Y**

Función:

(Y) − DATOS → NZC:

+ (Y > DATOS)	=	− (Y < DATOS)
−01	011	−00

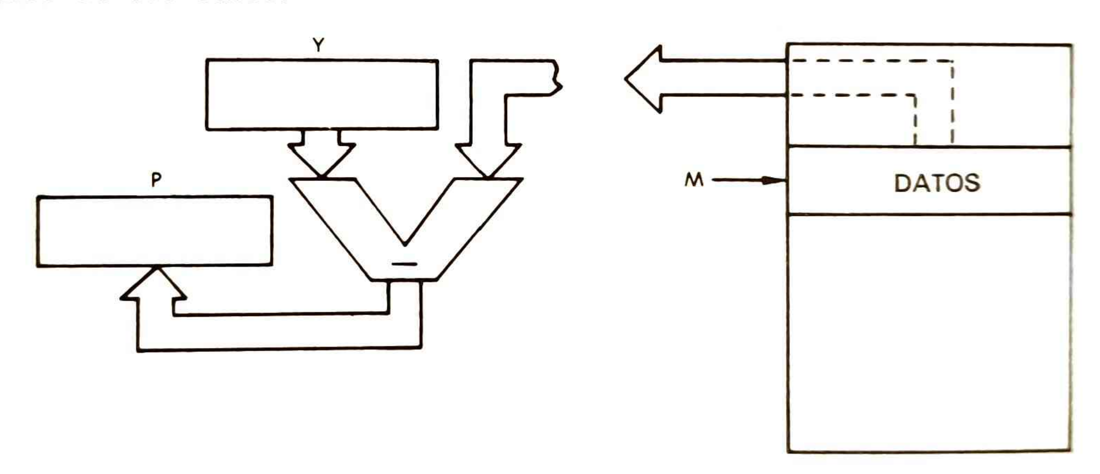
Formato:

1100bb00	DIRECCIÓN/DATOS	DIRECCIÓN

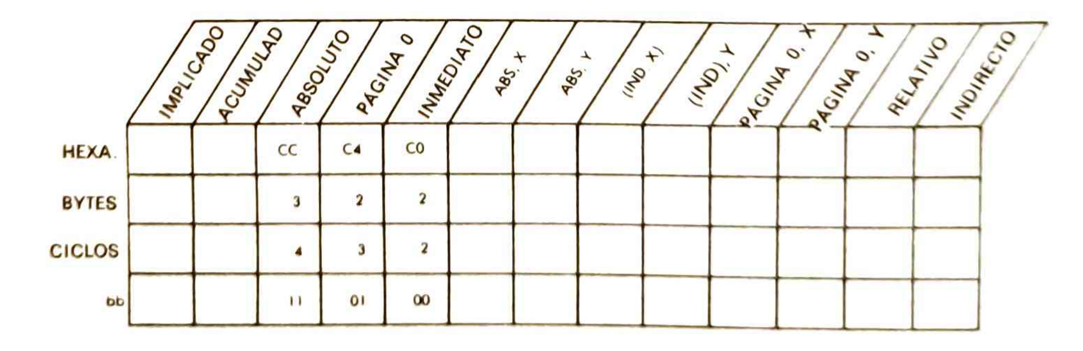
Descripción:

El contenido indicado se resta de Y. El resultado no se almacena, sino que los indicadores NZC se posicionan según que el resultado sea positivo, nulo o negativo. El valor del acumulador no cambia. CPY suele ir seguida por una bifurcación: BCC detecta Y < DATOS, BEQ detecta Y = DATOS y BEQ seguida por BCS detecta Y > DATOS. BCS detecta Y ⩾ DATOS.

Caminos de los datos:

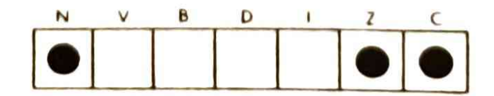


Modos de direccionamiento:



	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS. X	ABS. Y	(IND. X)	(IND), Y	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA.			CC	C4	C0								
BYTES			3	2	2								
CICLOS			4	3	2								
bb			11	01	00								

Indicadores:



Códigos de instrucción:

ABSOLUTO	11001100	DIRECCIÓN DE 16 BITS	
	bb = 11	HEXA. = CC	CICLOS = 4
PÁGINA-CERO	11000100	DIRECCIÓN	
	bb = 01	HEXA. = C4	CICLOS = 3
INMEDIATO	11000000	DATOS	
	bb = 00	HEXA. = CO	CICLOS = 2

DEC

Decrementar

Función:

M ← (M) − 1

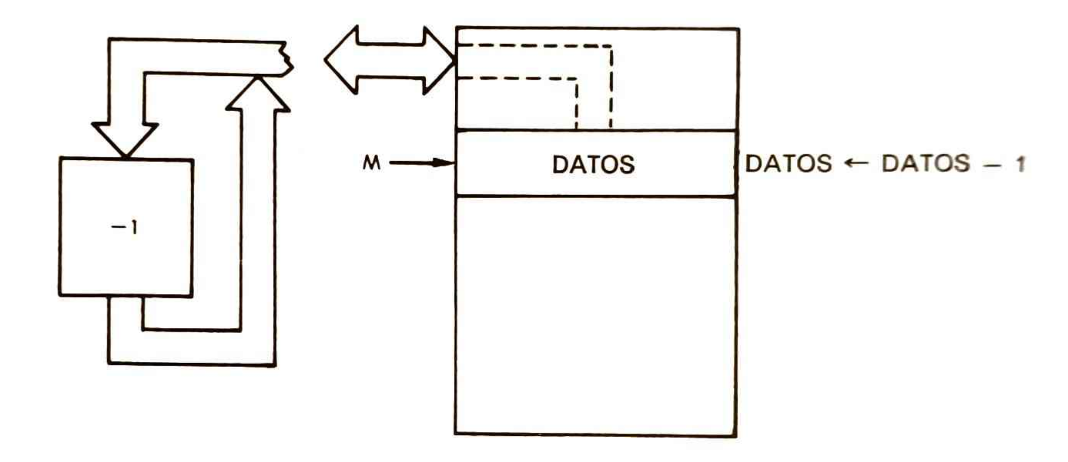
Formato:

110bb110	DIRECCIÓN	DIRECCIÓN

Descripción:

El contenido de la dirección de memoria indicada se disminuye en 1. El resultado se almacena de nuevo en la dirección de memoria indicada.

Caminos de los datos:



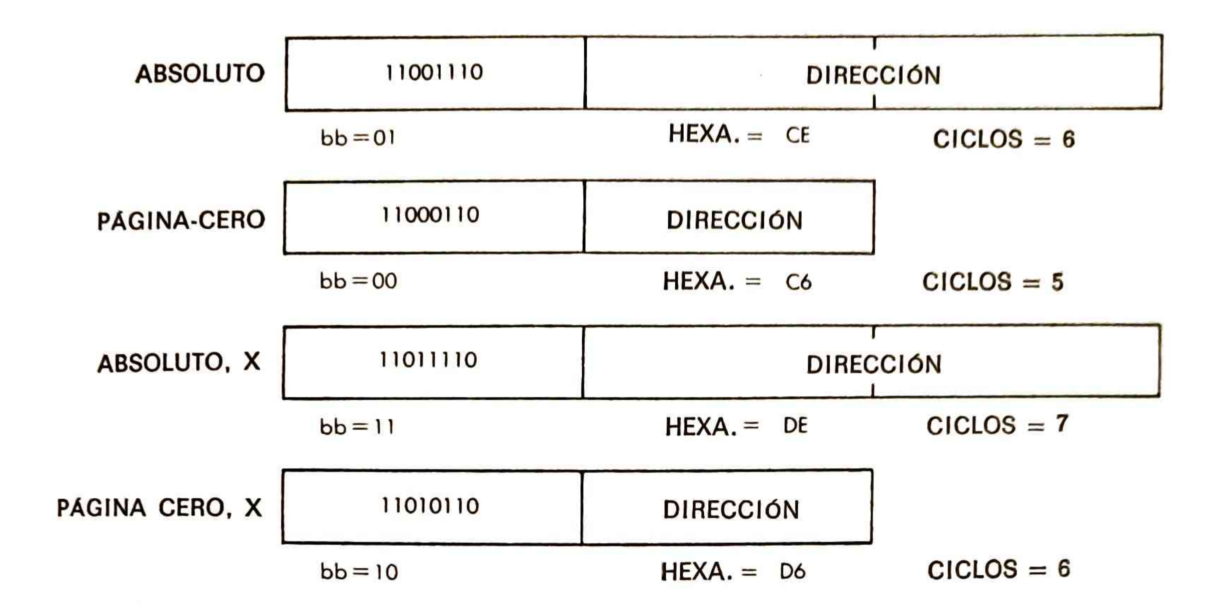
Modos de direccionamiento:

	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS, X	ABS, Y	(IND, X)	(IND), Y	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA.			CE	C6		DE				D6			
BYTES			3	2		3				2			
CICLOS			6	5		7				6			
bb			01	00		11				10			

Indicadores:

N	V	B	D	I	Z	C
●					●	

Códigos de instrucción:



DEX

Decrementar X

Función:

X ← (X) − 1

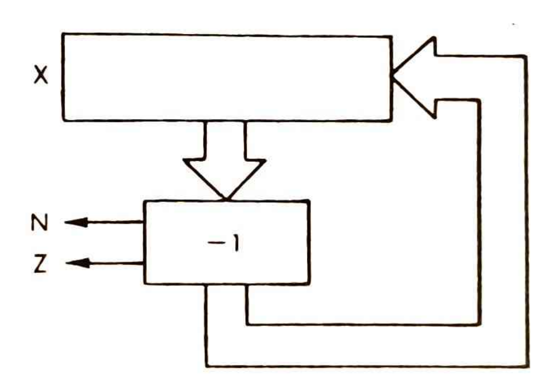
Formato:

11001010

Descripción:

El contenido de X se disminuye en 1. Permite utilizar X como contador.

Caminos de los datos:

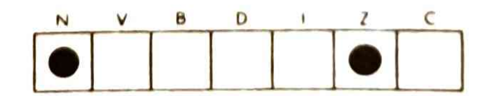


Modo de direccionamiento:

Solamente implícito:

HEX = CA, byte = 1, ciclos = 2.

Indicadores:



DEY   Decrementar Y

Función:

Y ← (Y) −1

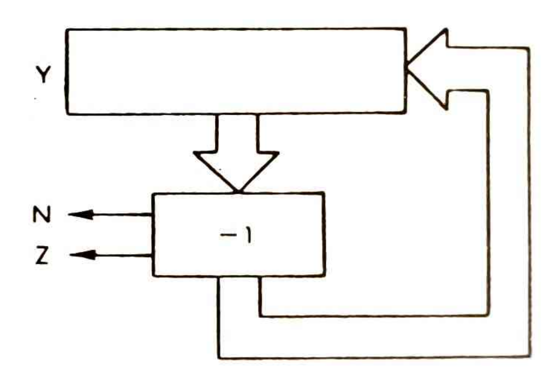
Formato:

10001000

Descripción:

El contenido de Y se disminuye en 1. Permite utilizar Y como contador.

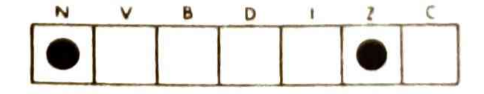
Caminos de los datos:



Modo de direccionamiento:

Solamente implícito:
HEX = 88, byte = 1, ciclos = 2.

Indicadores:



EOR **OR exclusiva con acumulador**

Función:

A ← (A) ⊻ DATOS

Formato:

010bbb01	DIRECCIÓN/DATOS	DIRECCIÓN

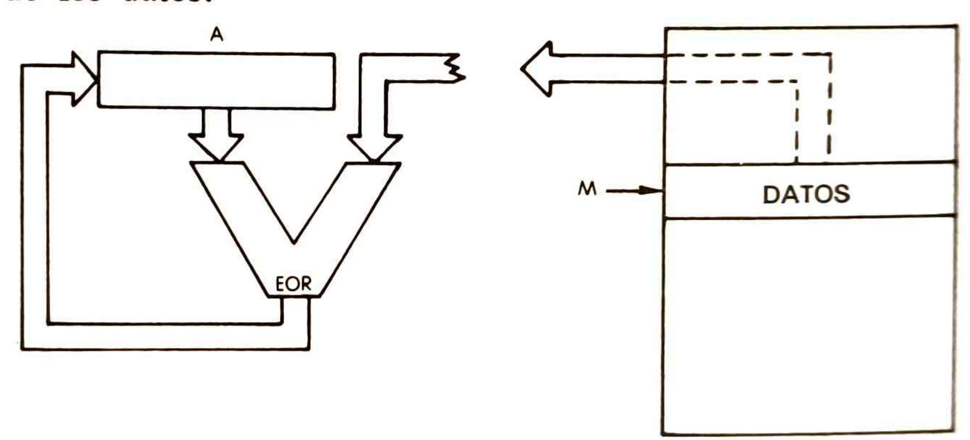
Descripción:

Se efectúa la operación OR exclusiva del contenido del acumulador y el dato indicado. La tabla de verdad es:

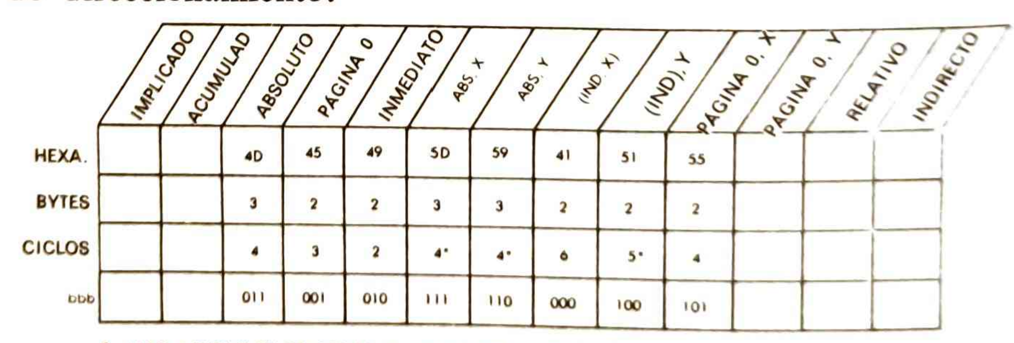
	0	1
0	0	1
1	1	0

Nota: EOR con "1" se puede utilizar para complementar.

Caminos de los datos:



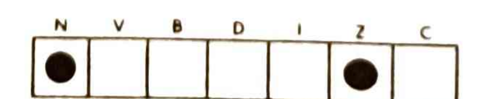
Modos de direccionamiento:



	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS. X	ABS. Y	(IND. X)	(IND), Y	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA.			4D	45	49	5D	59	41	51	55			
BYTES			3	2	2	3	3	2	2	2			
CICLOS			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

*: MAS 1 CICLO SI SE CRUZA EL LIMITE DE LA PAGINA.

Indicadores:



Códigos de instrucción:

Modo	Código	Operando	bbb	HEXA.	CICLOS
ABSOLUTO	01001101	DIRECCIÓN DE 16 BITS	011	4D	4
PÁGINA-CERO	01000101	DIRECCIÓN	001	45	3
INMEDIATO	01001001	DATOS	010	49	2
ABSOLUTO, X	01011101	DIRECCIÓN DE 16 BITS	111	5D	4*
ABSOLUTO, Y	01011001	DIRECCIÓN DE 16 BITS	110	59	4*
(IND, X)	01000001	DIRECCIÓN	000	41	6
(IND), Y	01010001	DIRECCIÓN	100	51	5*
PÁGINA CERO, X	01010101	DIRECCIÓN	101	55	4

*: MÁS 1 CICLO SI SE CRUZA EL LÍMITE DE LA PÁGINA.

INC

Incrementar memoria

Función:

M ← (M) +1

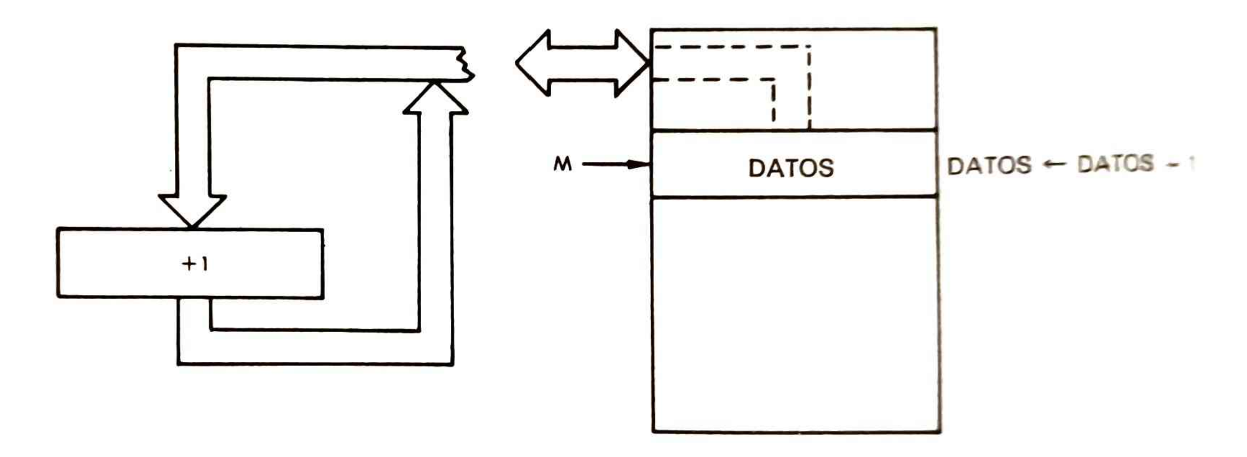
Formato:

111bb110	DIRECCIÓN	DIRECCIÓN

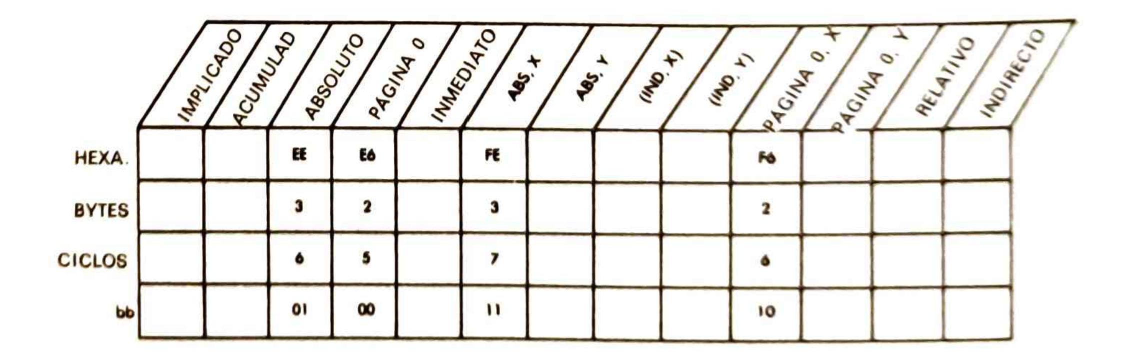
Descripción:

El contenido de la dirección de memoria indicada se aumenta en uno y, después, se vuelve a depositar en la misma dirección.

Caminos de los datos:

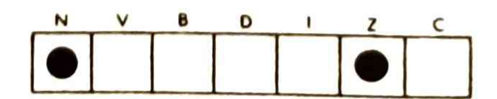


Modos de direccionamiento:

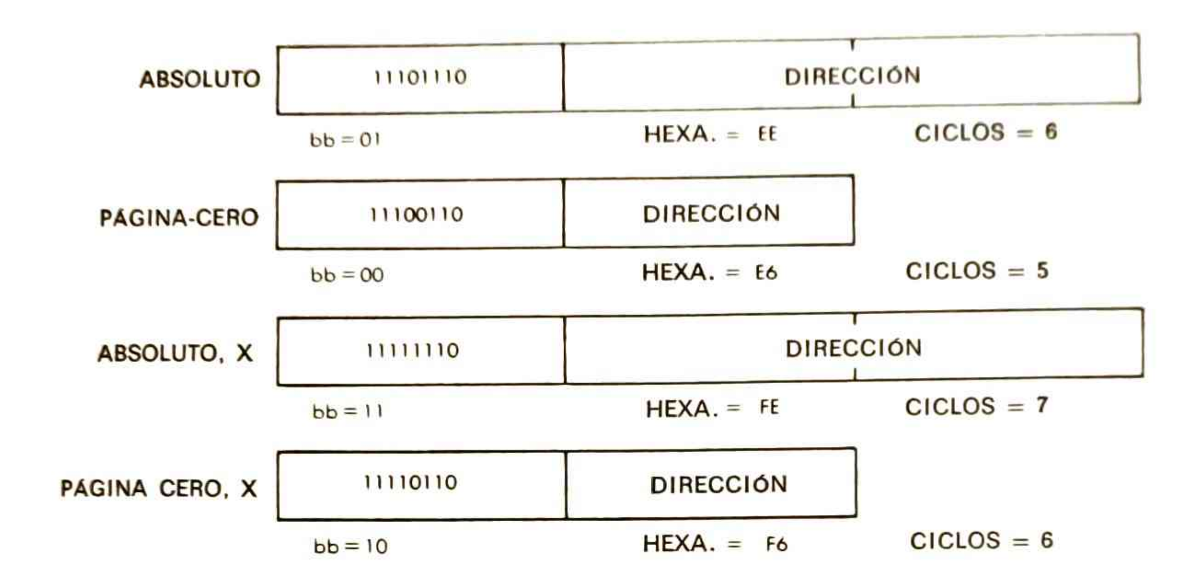


	IMPLICADO	ACUMULAD	ABSOLUTO	PÁGINA 0	INMEDIATO	ABS. X	ABS. Y	(IND. X)	(IND. Y)	PÁGINA 0, X	PÁGINA 0, Y	RELATIVO	INDIRECTO
HEXA.			EE	E6		FE				F6			
BYTES			3	2		3				2			
CICLOS			6	5		7				6			
bb			01	00		11				10			

Indicadores:



Códigos de instrucción:

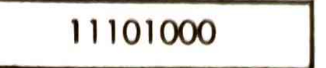


INX **Incrementar X**

Función:

X ← (X) + 1

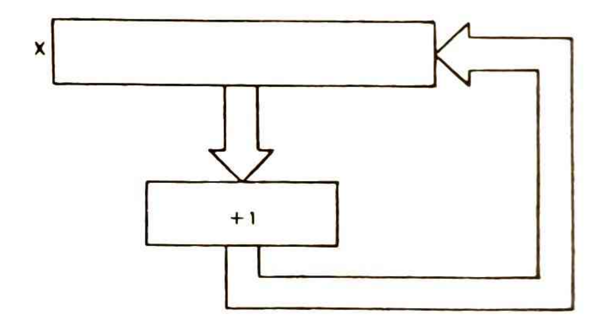
Formato:



Descripción:

El contenido de X se aumenta en uno. Ello permite utilizar a X como contador.

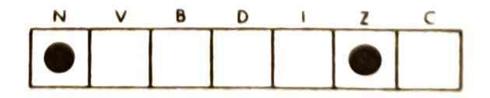
Caminos de los datos:



Modo de direccionamiento:

Solamente implícito:
HEX = E8, byte = 1, ciclos = 2.

Indicadores:



INY Incrementar Y

Función:

$Y \leftarrow (Y) + 1$

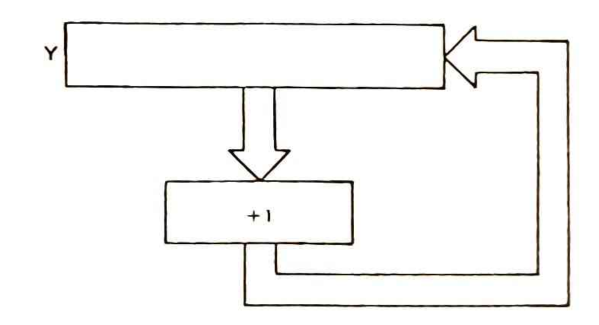
Formato:

11001000

Descripción:

El contenido de Y se incrementa en uno. Esto permite utilizar Y como contador.

Caminos de los datos:

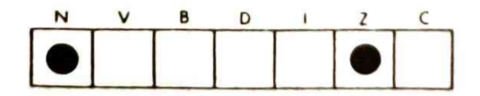


Modo de direccionamiento:

Solamente implícito:

HEX = C8, byte = 1, ciclos = 2.

Indicadores:



JMP

Salto a una dirección

Función:

PC ← DIRECCIÓN

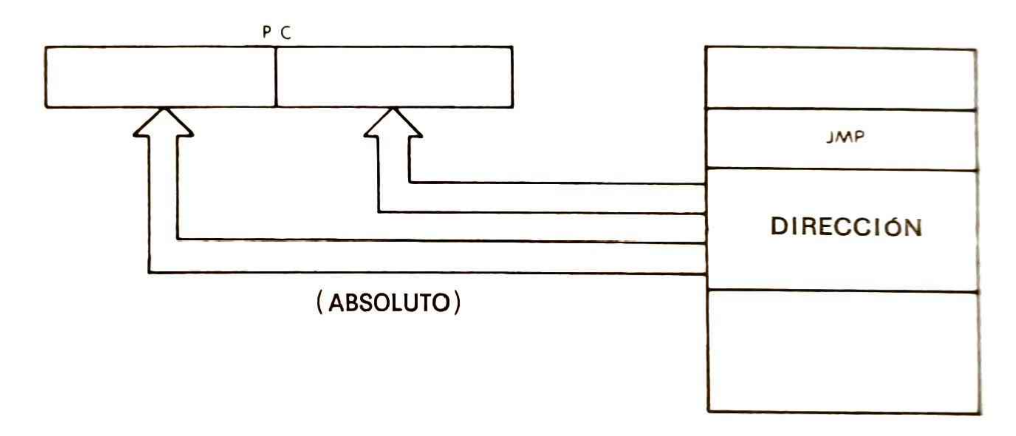
Formato:

01001100	DIRECCIÓN

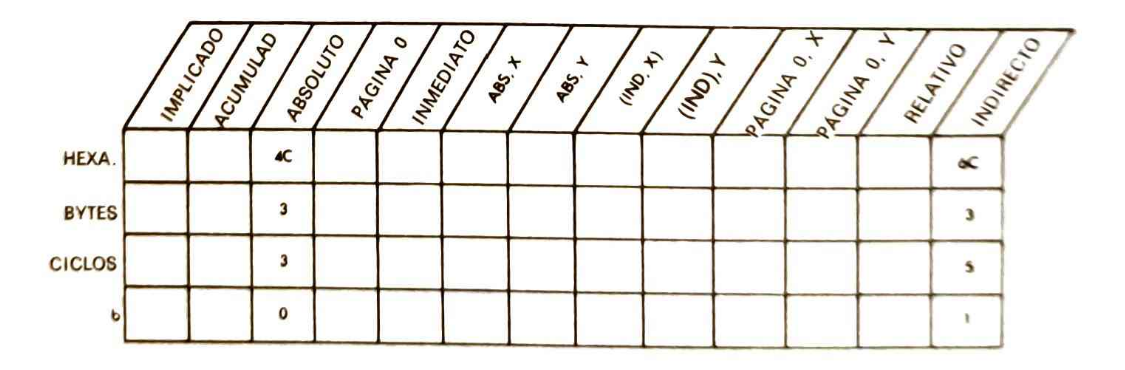
Descripción:

Se carga en el contador de programa una nueva dirección, produciendo un salto en la secuencia del programa. La especificación de la dirección puede ser absoluta o indirecta.

Caminos de los datos:



Modos de direccionamiento:

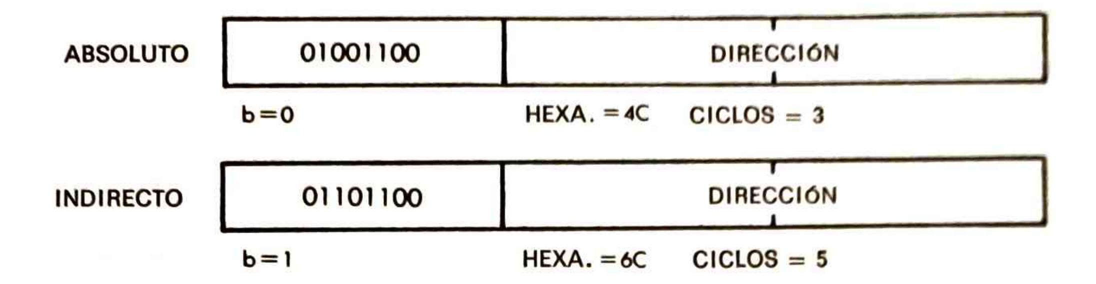


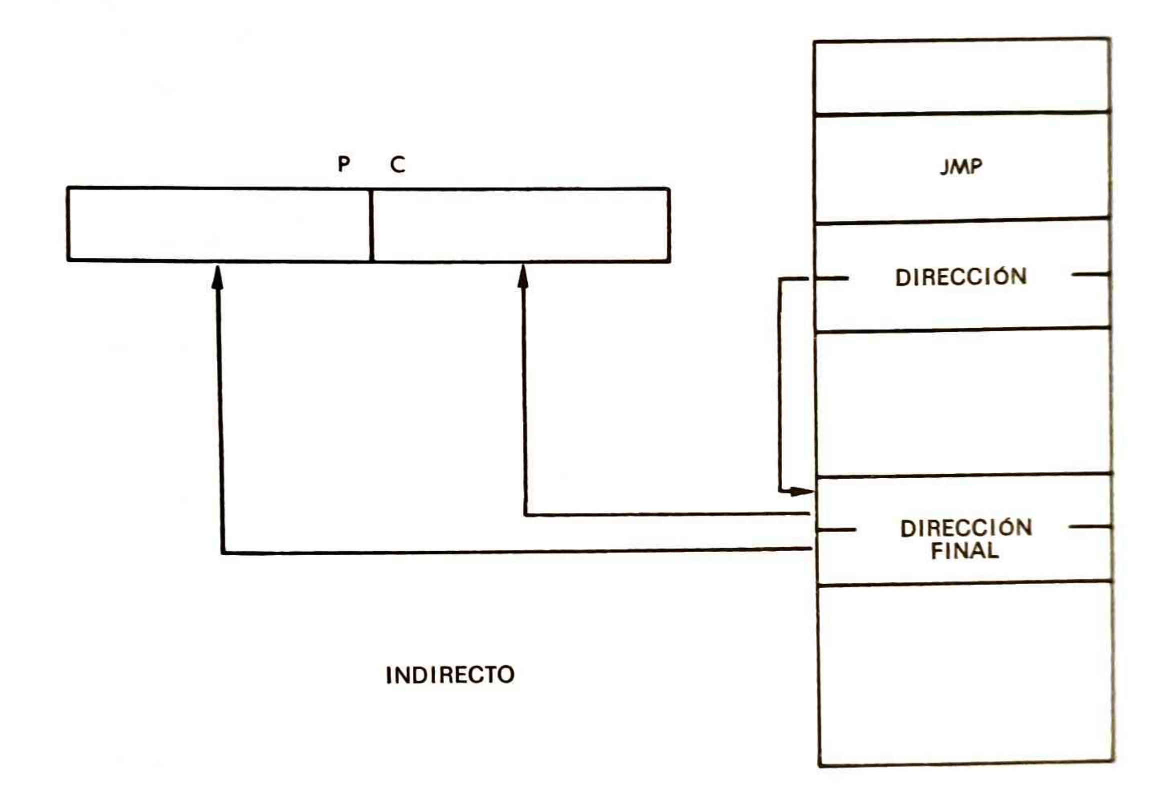
	IMPLICADO	ACUMULAD	ABSOLUTO	PÁGINA 0	INMEDIATO	ABS, X	ABS, Y	(IND, X)	(IND), Y	PÁGINA 0, X	PÁGINA 0, Y	RELATIVO	INDIRECTO
HEXA.			4C										6C
BYTES			3										3
CICLOS			3										5
b			0										1

Indicadores:



Códigos de instrucción:





INDIRECTO

JSR

Salto a subrutina

Función:

PILA ← (PC) + 2
PC ← DIRECCIÓN

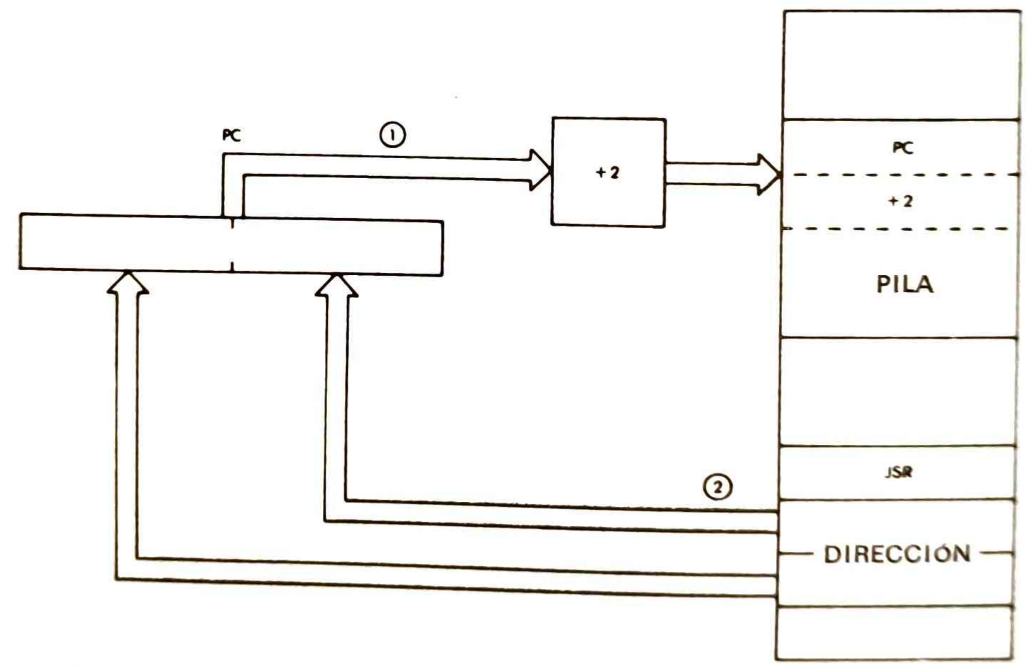
Formato:



Descripción:

El contenido del contador de programa +2 se guarda en la pila. (Es la dirección de la instrucción siguiente a JSR.) La dirección de la subrutina se carga a continuación en el PC. Se llama también "LLAMADA a subrutina" (subroutine CALL).

Caminos de los datos:



Modo de direccionamiento:

Solamente absoluto:
HEX = 20, bytes = 3, ciclos = 6.

Indicadores:



LDA Cargar acumulador

Función:

A ← DATOS

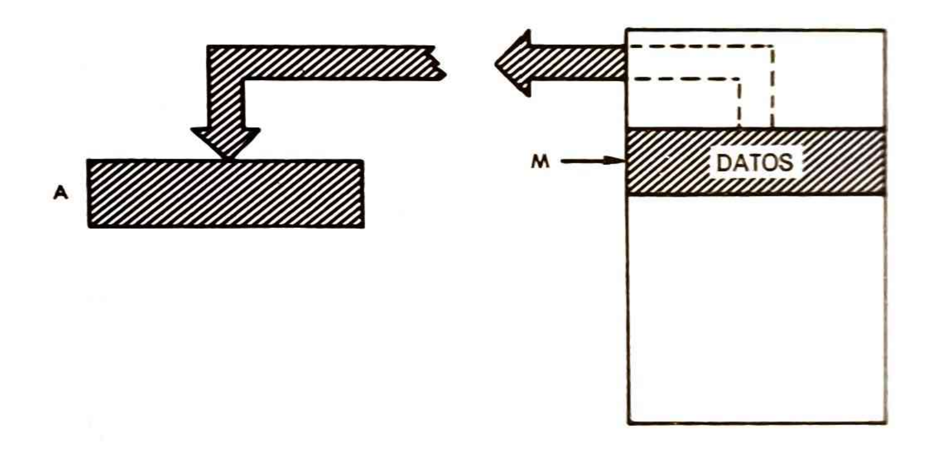
Formato:

101bbb01	DIRECCIÓN/DATOS	DIRECCIÓN

Descripción:

El acumulador se carga con nuevos datos.

Caminos de los datos:

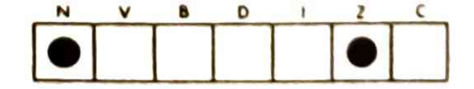


Modos de direccionamiento:

	IMPLICADO	ACUMULAD	ABSOLUTO	PÁGINA 0	INMEDIATO	ABS. X	ABS. Y	(IND, X)	(IND), Y	PÁGINA 0, X	PÁGINA 0, Y	RELATIVO	INDIRECTO
HEXA.			AD	A5	A9	BD	B9	A1	B1	B5			
BYTES			3	2	2	3	3	2	2	2			
CICLOS			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

*: MÁS 1 CICLO SI SE CRUZA EL LIMITE DE LA PÁGINA.

Indicadores:



Códigos de instrucción:

Modo	Código	Operando	bbb	HEXA.	CICLOS
ABSOLUTO	10101101	DIRECCIÓN DE 16 BITS	bbb = 011	HEXA. = AD	CICLOS = 4
PAGINA-CERO	10100101	DIRECCIÓN	bbb = 001	HEXA. = A5	CICLOS = 3
INMEDIATO	10101001	DATOS	bbb = 010	HEXA. = A9	CICLOS = 2
ABSOLUTO, X	10111101	DIRECCIÓN DE 16 BITS	bbb = 111	HEXA. = BD	CICLOS = 4*
ABSOLUTO, Y	10111001	DIRECCIÓN DE 16 BITS	bbb = 110	HEXA. = B9	CICLOS = 4*
(IND, X)	10100001	DIRECCIÓN	bbb = 000	HEXA. = A1	CICLOS = 6
(IND),Y	10110001	DIRECCIÓN	bbb = 100	HEXA. = B1	CICLOS = 5*
PÁGINA CERO, X	10110101	DIRECCIÓN	bbb = 101	HEXA. = B5	CICLOS = 4

*: MÁS 1 CICLO SI SE CRUZA EL LÍMITE DE LA PÁGINA.

LDX   Cargar registro X

Función:

X ← DATOS

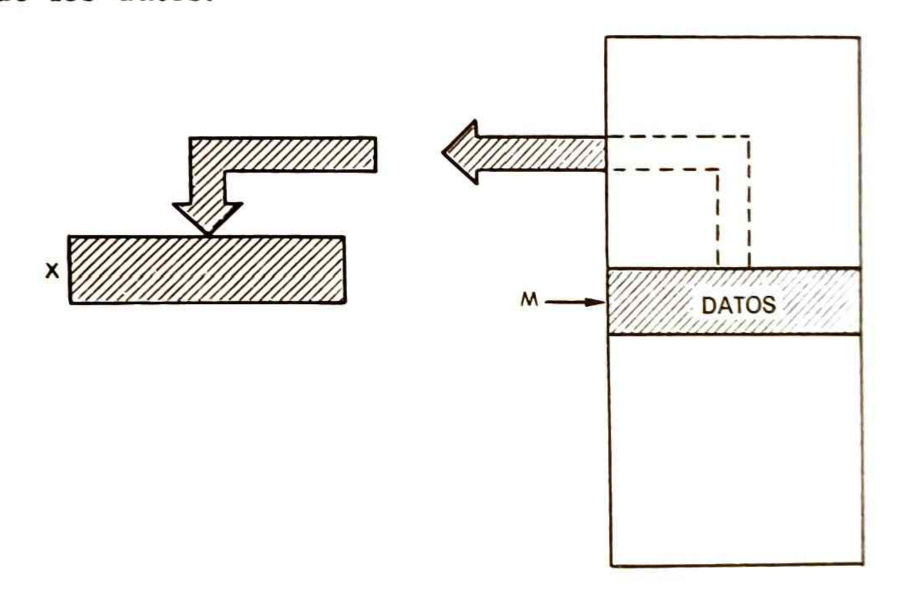
Formato:

101bbb10	DIRECCIÓN/DATOS	DIRECCIÓN

Descripción:

El registro índice X se carga con los datos contenidos en la dirección indicada.

Caminos de los datos:



Modos de direccionamiento:

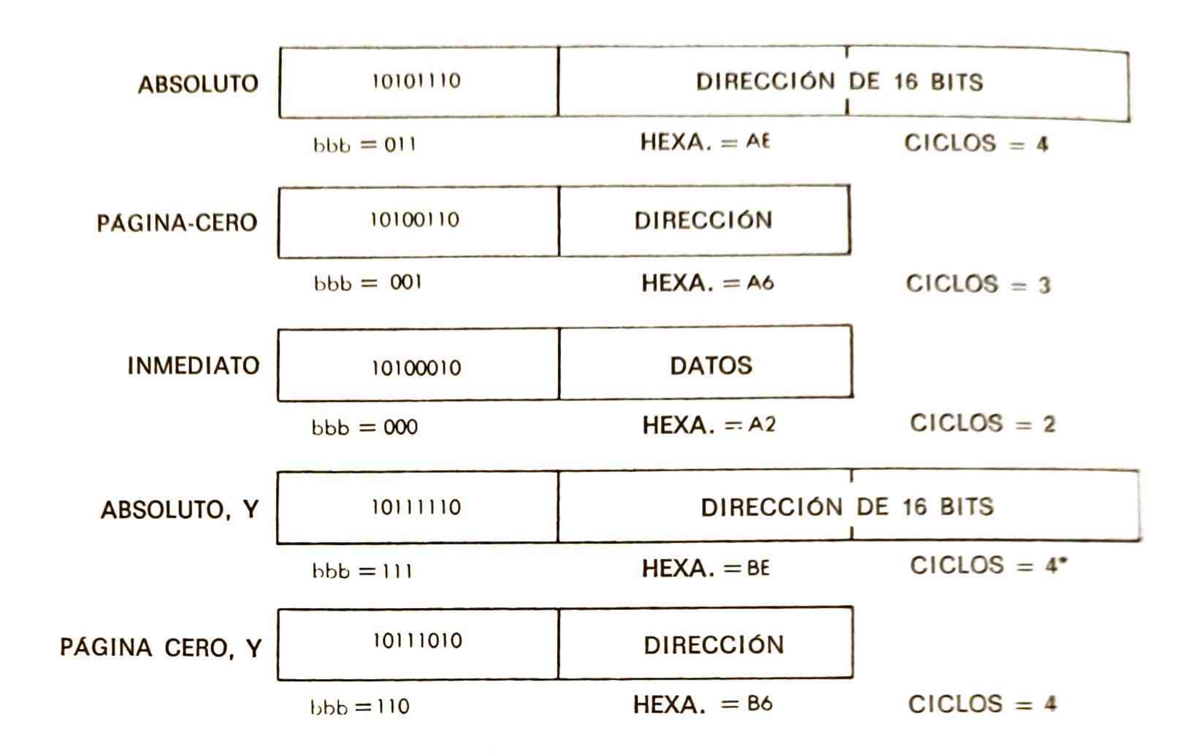
	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS. X	ABS. Y	(IND. X)	(IND). Y	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA.			AE	A6	A2		BE				B6		
BYTES			3	2	2		3				2		
CICLOS			4	3	2		4*				4		
bbb			011	001	000		111				110		

*: MÁS 1 CICLO SI SE CRUZA EL LIMITE DE LA PAGINA

Indicadores:

N	V	B	D	I	Z	C
●					●	

Códigos de instrucción:



*: MÁS 1 CICLO SI SE CRUZA EL LÍMITE DE LA PÁGINA.

LDY **Cargar registro Y**

Función:

Y ← DATOS

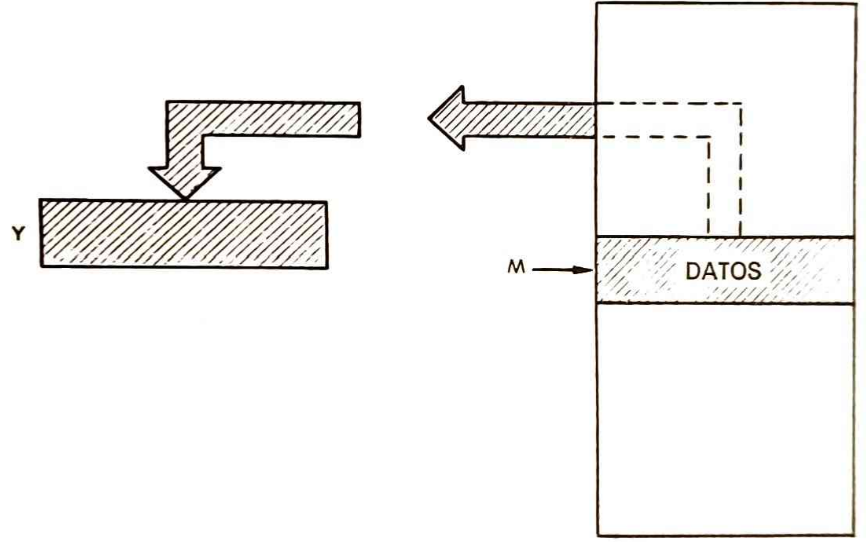
Formato:

101bbb00	DIRECCIÓN/DATOS	DIRECCIÓN

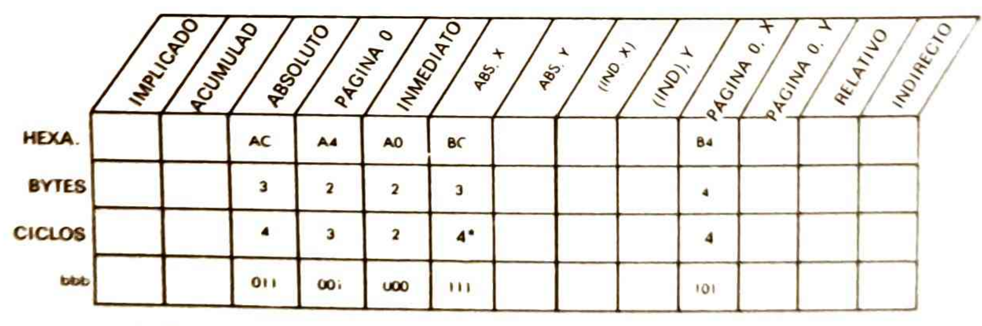
Descripción:

El registro índice Y se carga con los datos contenidos en la dirección indicada.

Caminos de los datos:



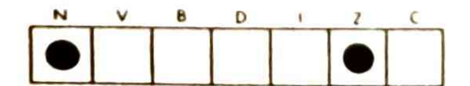
Modos de direccionamiento:



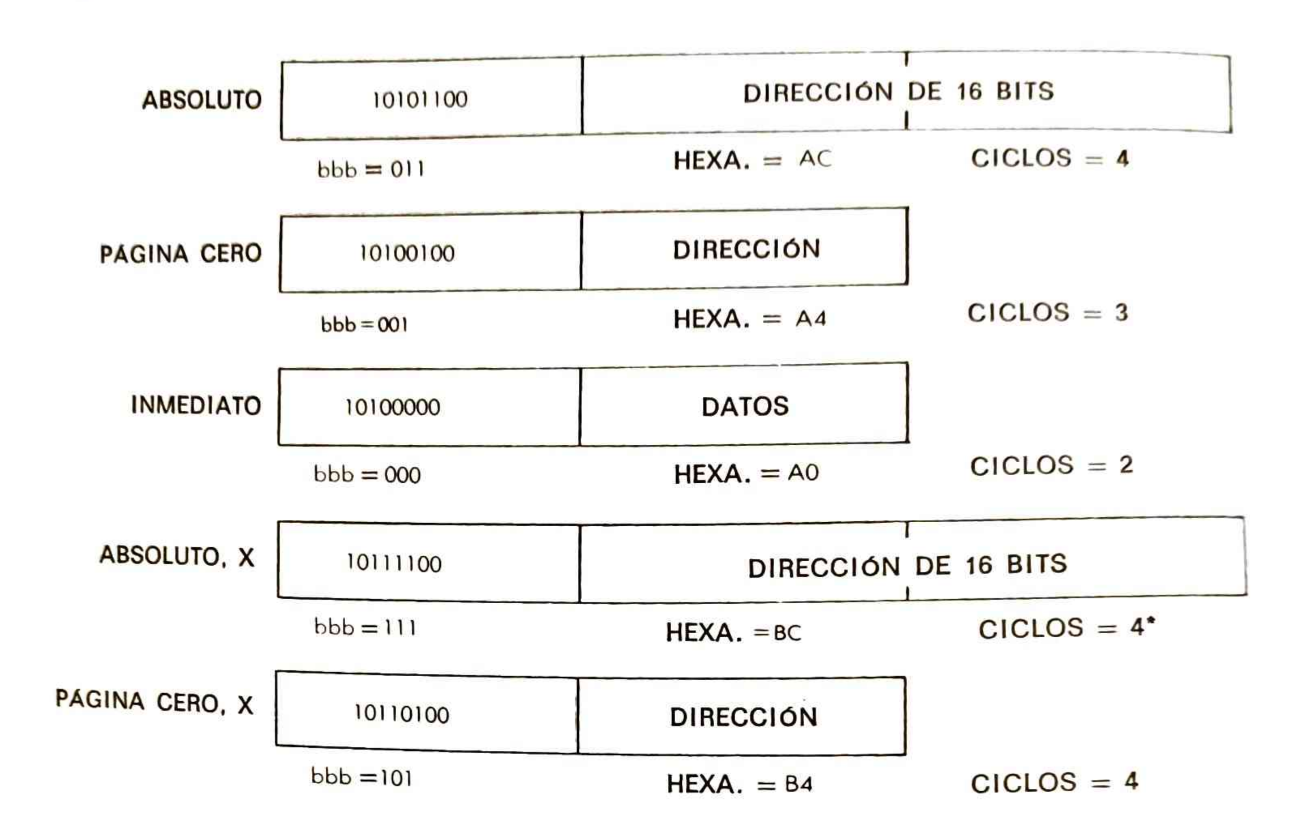
	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS. X	ABS. Y	(IND X)	(IND), Y	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA.			AC	A4	A0	BC				B4			
BYTES			3	2	2	3				4			
CICLOS			4	3	2	4*				4			
bbb			011	001	000	111				101			

*: MAS 1 CICLO SI SE CRUZA EL LIMITE DE LA PAGINA

Indicadores:



Códigos de instrucción:



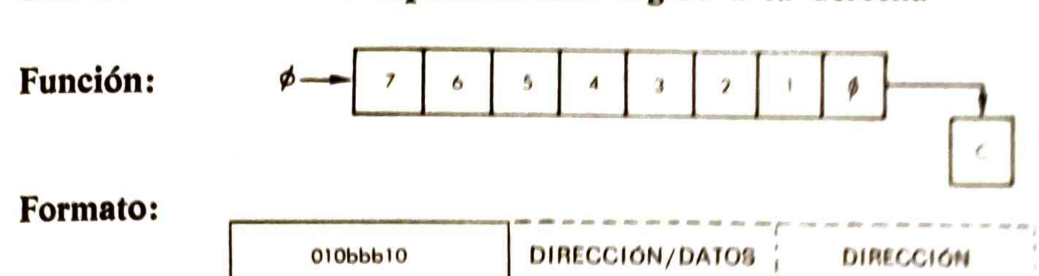
*: MÁS 1 CICLO SI SE CRUZA EL LÍMITE DE LA PÁGINA.



LSR

Desplazamiento lógico a la derecha

Función:

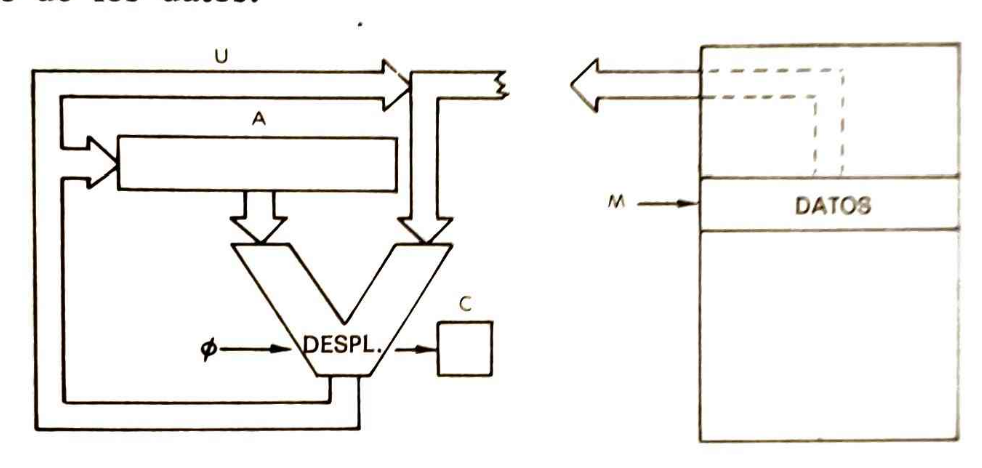


Formato:

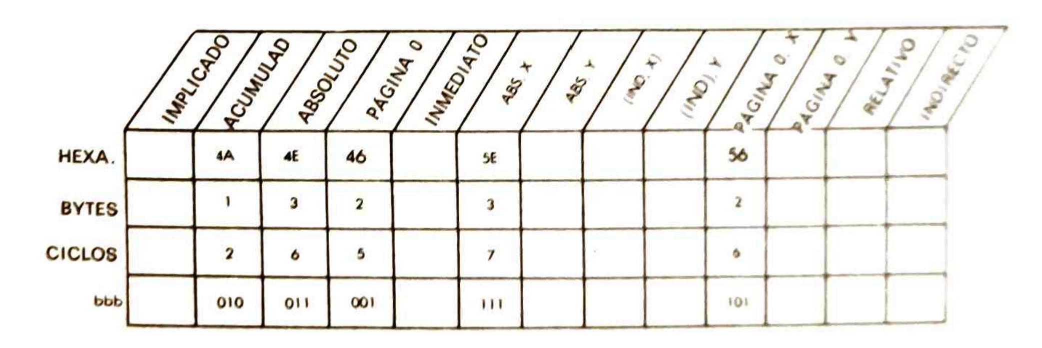
Descripción:

Desplaza en una posición de bit a la derecha el contenido indicado (acumulador o memoria). Se fuerza un "0" en el bit 7. El bit 0 se transfiere al acarreo. Los datos desplazados se depositan en la fuente, es decir, en el acumulador o en la memoria.

Caminos de los datos:

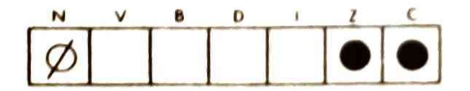


Modos de direccionamiento:

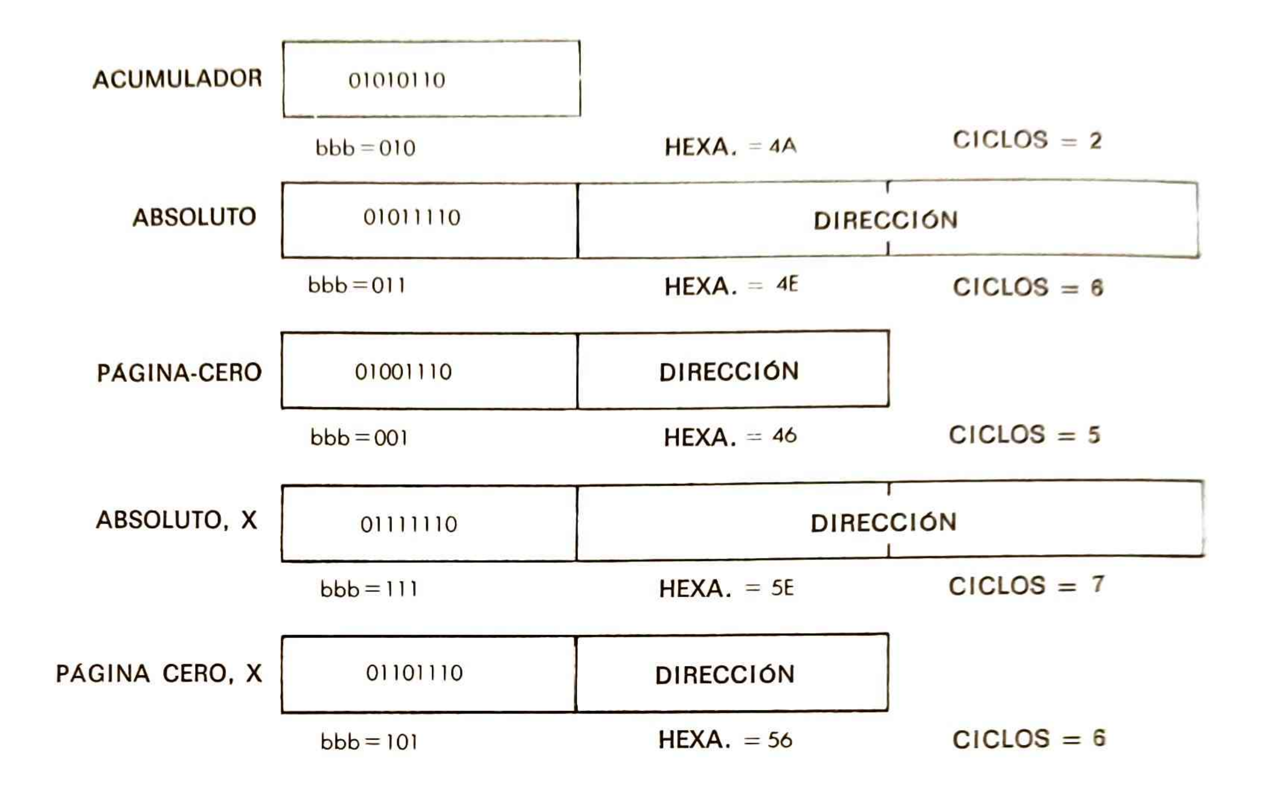


	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS, X	ABS, Y	(IND, X)	(IND), Y	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA.		4A	4E	46		5E				56			
BYTES		1	3	2		3				2			
CICLOS		2	6	5		7				6			
bbb		010	011	001		111				101			

Indicadores:



Códigos de instrucción:



NOP

Ninguna operación

Función:

Ninguna.

Formato:

11101010

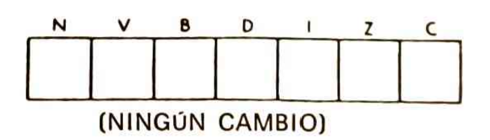
Descripción:

No hace nada durante 2 ciclos. Puede servir para temporizar un bucle de retardo o para rellenar "parches" en un programa.

Modo de direccionamiento:

Solamente implícito:
HEX = EA, byte = 1, ciclos = 2.

Indicadores:



ORA

OR inclusiva con acumulador

Función:

A ← (A) V DATOS

Formato:

000bbb01	DIRECCIÓN/DATOS

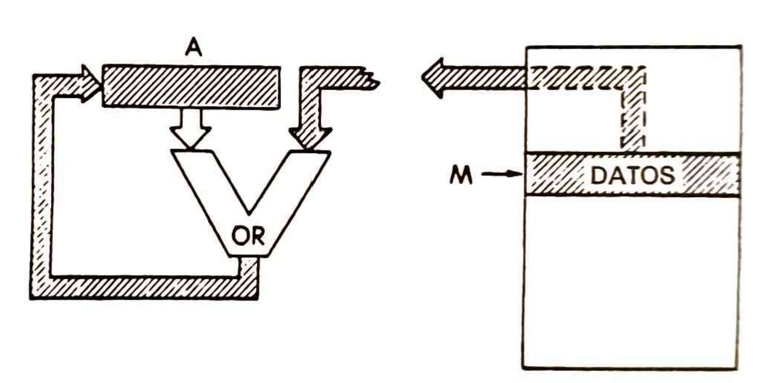
Descripción:

Efectúa la operación OR (inclusiva) de A y del dato indicado. El resultado se almacena en A. Puede ser utilizado para forzar a "1" en las posiciones de bits seleccionadas.

Tabla de verdad:

	0	1
0	0	1
1	1	1

Caminos de los datos:

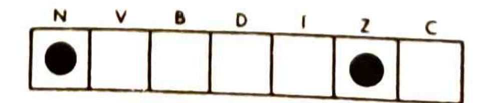


Modos de direccionamiento:

	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS, X	ABS, Y	(IND, X)	(IND), Y	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA.			0D	05	09	1D	19	01	11	15			
BYTES			3	2	2	3	3	2	2	2			
CICLOS			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

*: MAS 1 CICLO SI SE CRUZA EL LIMITE DE LA PAGINA.

Indicadores:



Códigos de instrucción:

ABSOLUTO	00001101	DIRECCIÓN DE 16 BITS	bbb = 011	HEXA. = 0D	CICLOS = 4
PAGINA-CERO	00000101	DIRECCIÓN	bbb = 001	HEXA. = 05	CICLOS = 3
INMEDIATO	00001001	DATOS	bbb = 010	HEXA. = 09	CICLOS = 2
ABSOLUTO, X	00011101	DIRECCIÓN DE 16 BITS	bbb = 111	HEXA. = 1D	CICLOS = 4*
ABSOLUTO, Y	00011001	DIRECCIÓN DE 16 BITS	bbb = 110	HEXA. = 19	CICLOS = 4*
(IND, X)	00000001	DIRECCIÓN	bbb = 000	HEXA. = 01	CICLOS = 6
(IND),Y	00010001	DIRECCIÓN	bbb = 100	HEXA. = 11	CICLOS = 5*
PÁGINA CERO, X	00010101	DIRECCIÓN	bbb = 101	HEXA. = 15	CICLOS = 4

*: MÁS 1 CICLO SI SE CRUZA EL LÍMITE DE LA PÁGINA.

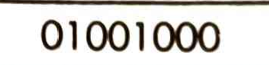
PHA

Introducir en pila A

Función:

PILA ← (A)
S ← (S) − 1

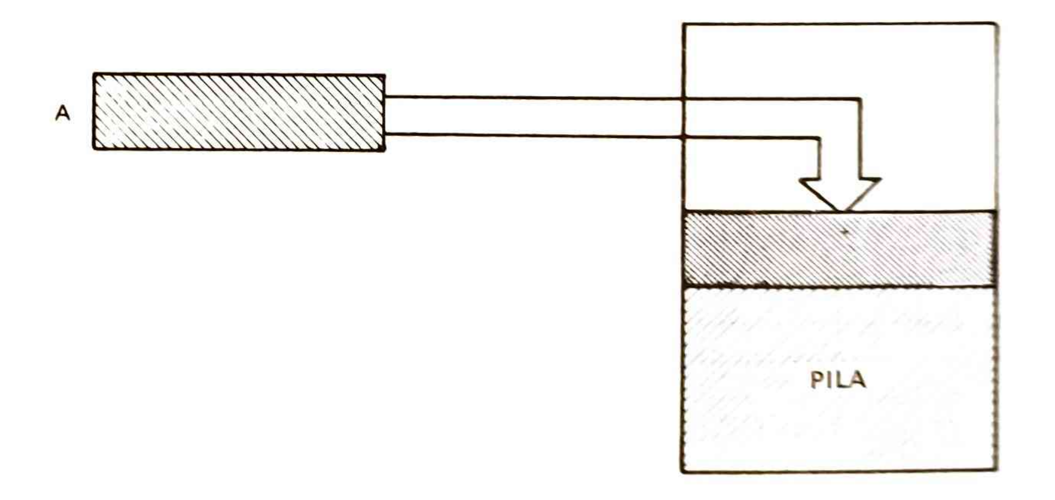
Formato:



Descripción:

El contenido del acumulador se introduce en la pila. El puntero de pila se actualiza. "A" no se altera.

Caminos de los datos:



Modo de direccionamiento:

Solamente implícito:
HEX = 48, byte = 1, ciclos = 3.

Indicadores:

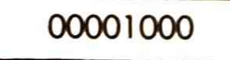


PHP Introducir en pila el estado del procesador

Función:

PILA ← (P)
S ← (S) − 1

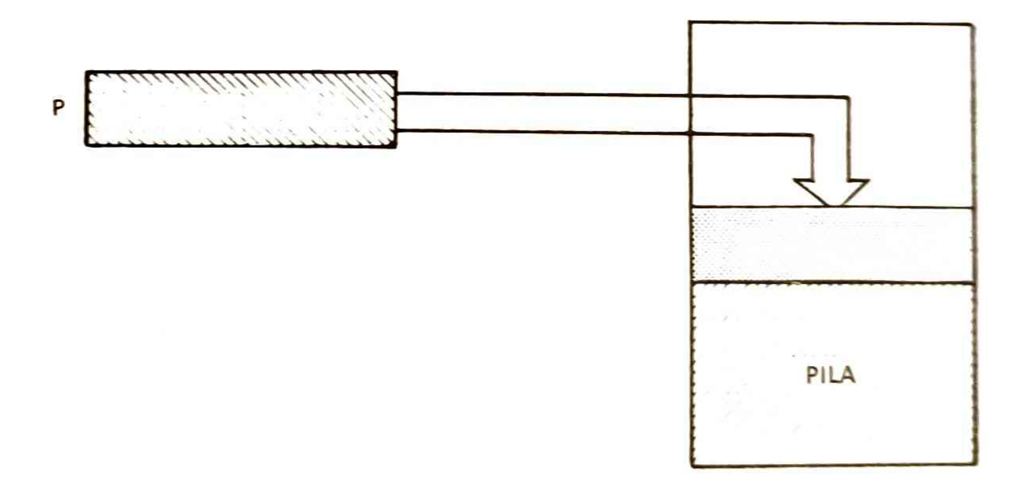
Formato:



Descripción:

El contenido del registro de estado P se introduce en la pila. El puntero de pila se actualiza. "A" no cambia.

Caminos de los datos:



Modo de direccionamiento:

Solamente implícito:
HEX =08, byte = 1, ciclos = 3.

Indicadores:



PLA

Extraer acumulador

Función:

A ← (PILA)
S ← (S) + 1

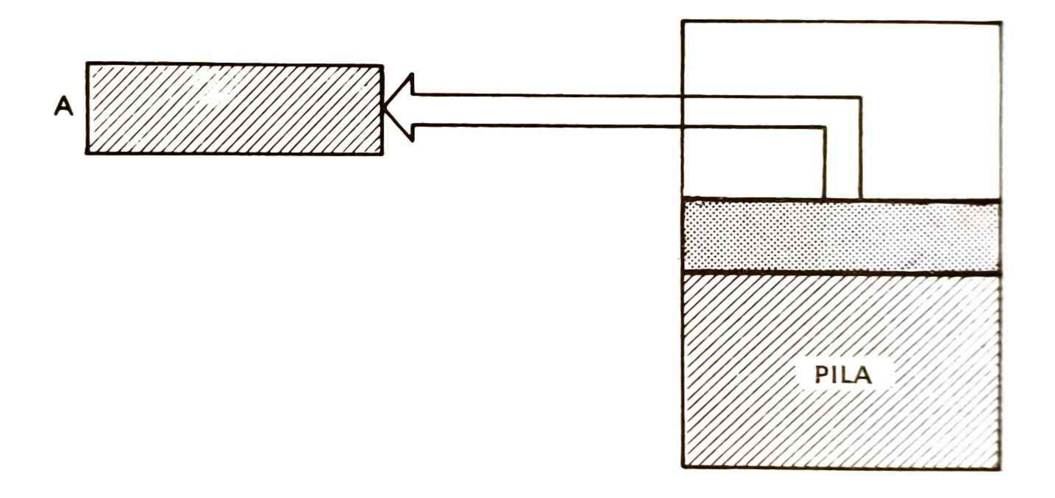
Formato:

01101000

Descripción:

Extrae la palabra de la cima de la pila y la transfiere al acumulador. El puntero de pila se incrementa.

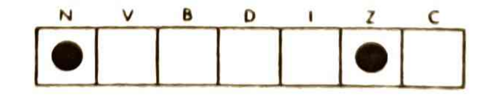
Caminos de los datos:



Modo de direccionamiento:

Solamente implícito:
HEX = 68, byte = 1, ciclos = 4.

Indicadores:



PLP **Extraer el estado del procesador desde la pila**

Función:

P ← (PILA)
S ← (S) + 1

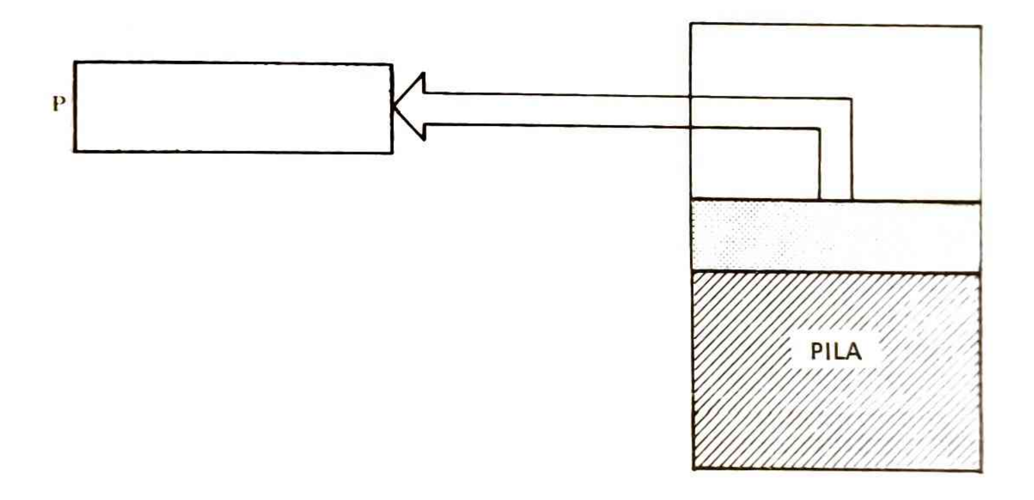
Formato:

00101000

Descripción:

La palabra de la cima de la pila se extrae (transfiere) al registro de estado P. El puntero de pila se incrementa.

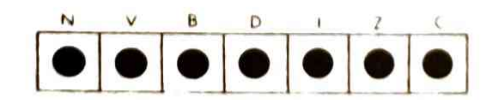
Caminos de los datos:



Modo de direccionamiento:

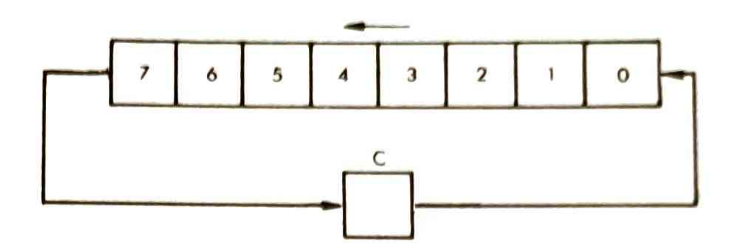
Solamente implícito:
HEX = 28, byte = 1, ciclos = 4.

Indicadores:



ROL Rotación de un bit a la izquierda

Función:



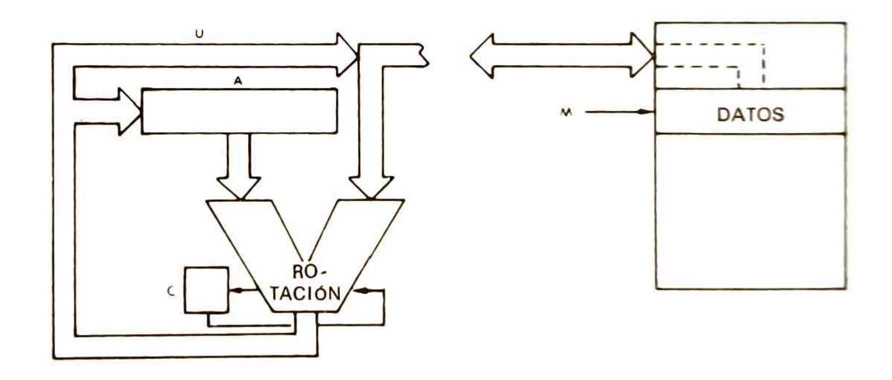
Formato:



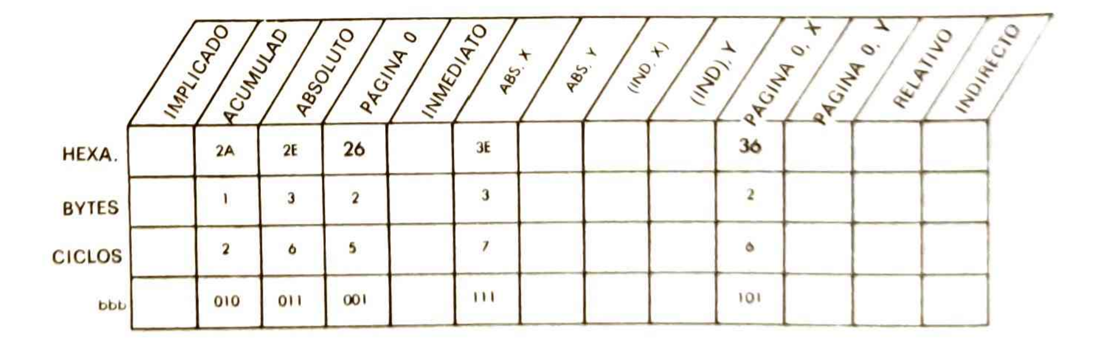
Descripción:

El contenido de la dirección indicada (acumulador o memoria) se rota una posición a la izquierda. El acarreo pasa al bit 0. El bit 7 se pone al nuevo valor del acarreo. Se trata, pues, de una rotación sobre 9 bits.

Caminos de los datos:

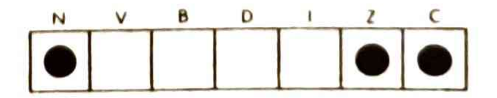


Modos de direccionamiento:

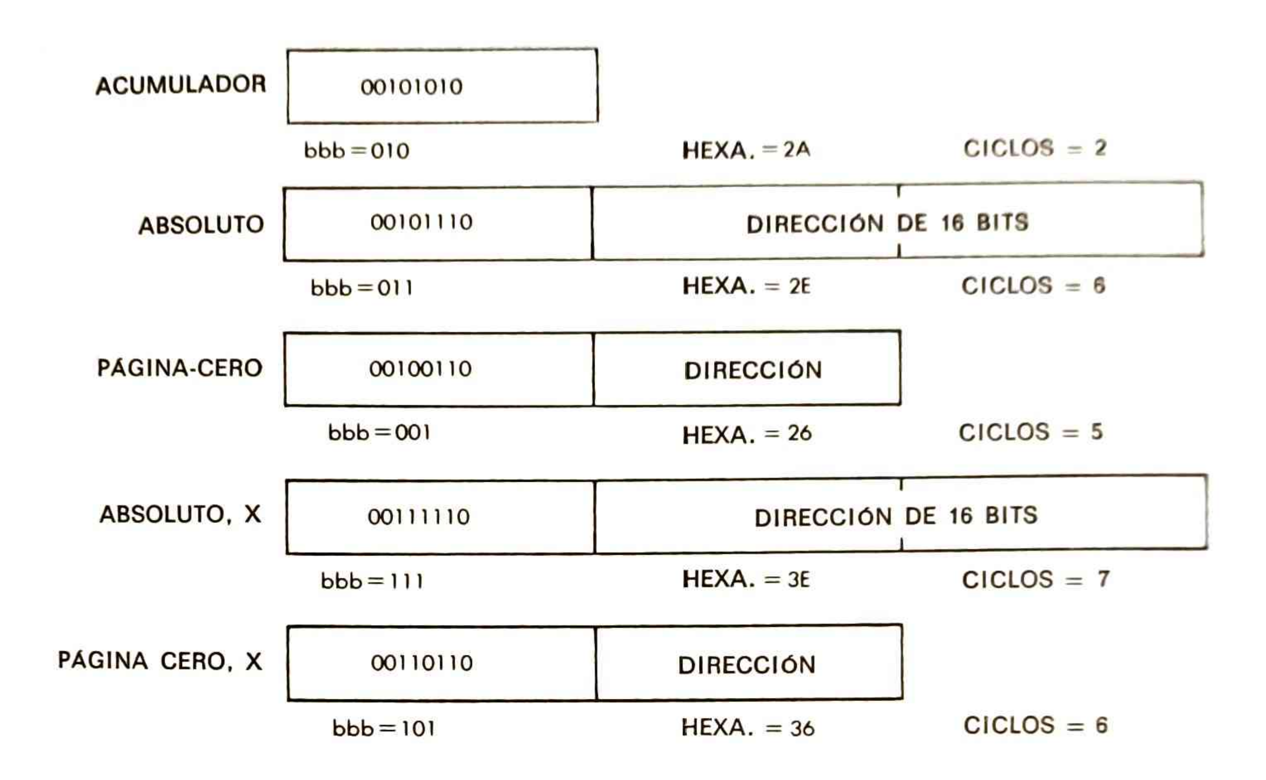


	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS. X	ABS. Y	(IND. X)	(IND), Y	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA.		2A	2E	26		3E				36			
BYTES		1	3	2		3				2			
CICLOS		2	6	5		7				6			
bbb		010	011	001		111				101			

Indicadores:



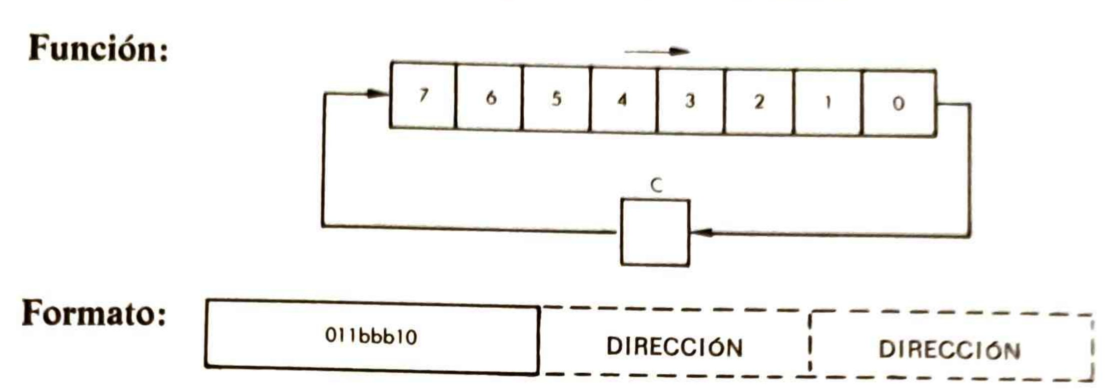
Códigos de instrucción:



ROR — Rotación de un bit a la derecha

Atención: Esta instrucción puede no estar disponible en los 6502 más antiguos; puede existir también, pero no estar listada.

Función:



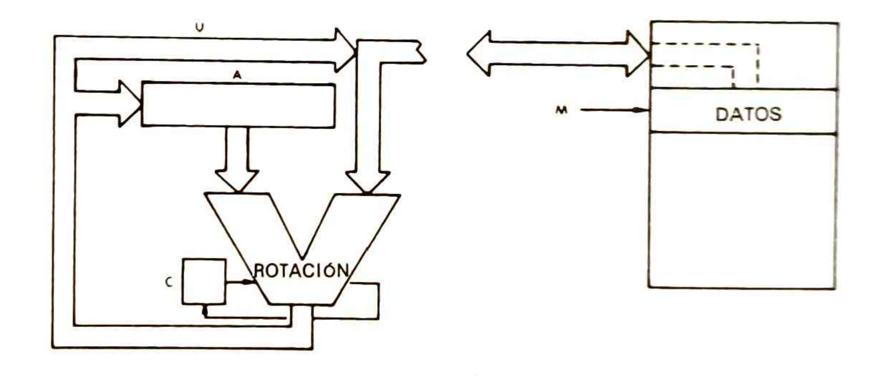
Formato:

011bbb10	DIRECCIÓN	DIRECCIÓN

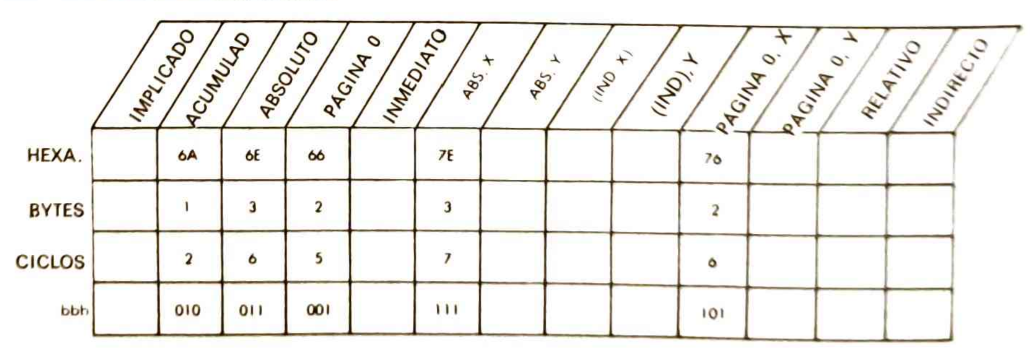
Descripción:

El contenido de la dirección indicada (acumulador o memoria) se rota a la derecha la posición de un bit. El acarreo va al bit 7. El bit 0 se pone al nuevo valor del acarreo. Es una rotación de 9 bits.

Caminos de los datos:

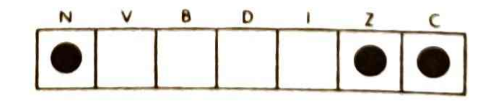


Modos de direccionamiento:

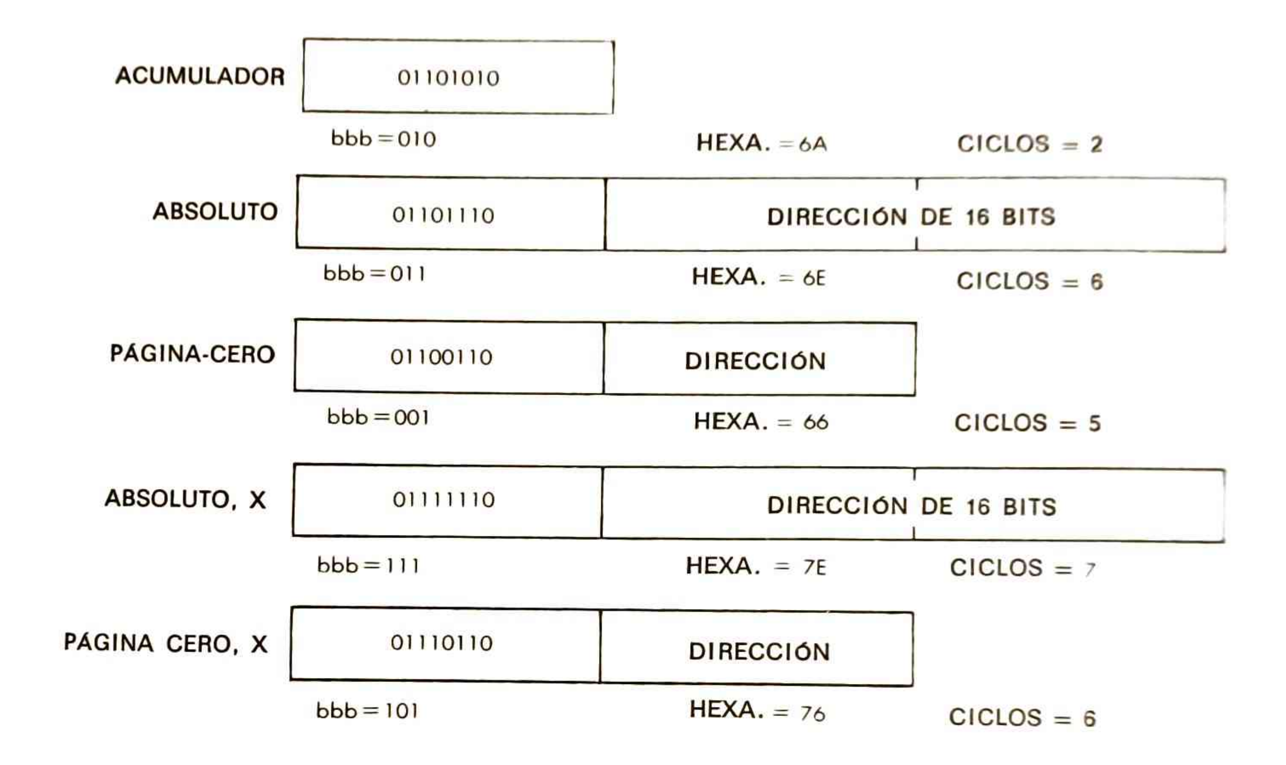


	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS. X	ABS. Y	(IND X)	(IND), Y	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA.		6A	6E	66		7E				76			
BYTES		1	3	2		3				2			
CICLOS		2	6	5		7				6			
bbb		010	011	001		111				101			

Indicadores:



Códigos de instrucción:



RTI

Retorno desde interrupción

Función:

P ← (PILA)
S ← (S) + 1
PCL ← (PILA)
S ← (S + 1)
PCH ← (PILA)
S ← (S + 1)

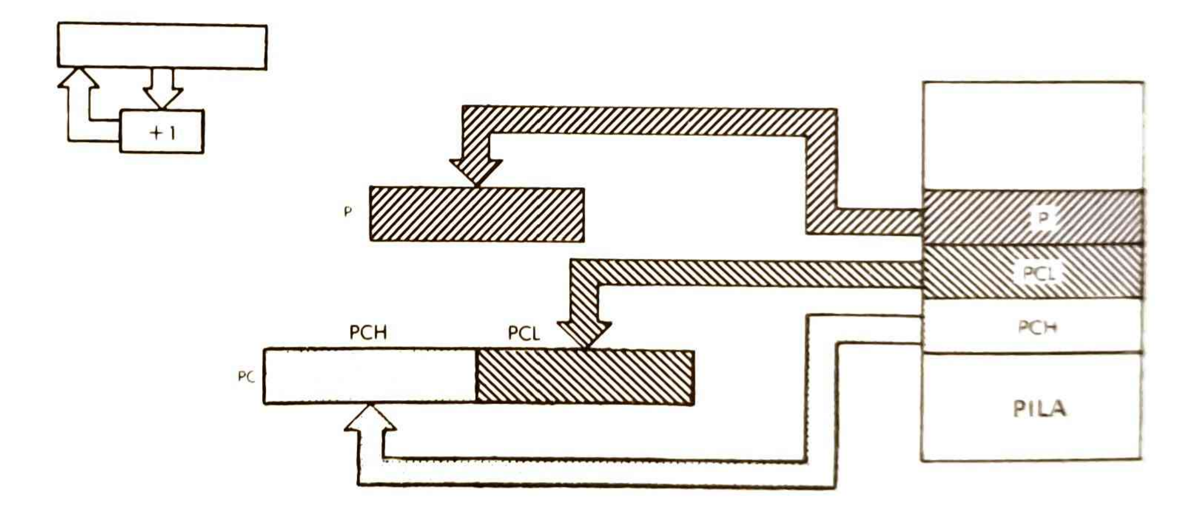
Formato:

01000000

Descripción:

Restaura el registro de estado P y el contador de programa (PC) que han sido guardados en la pila. Actualiza el puntero de pila.

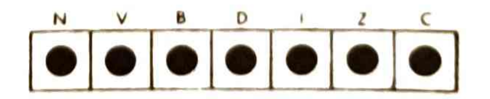
Caminos de los datos:



Modo de direccionamiento:

Solamente implícito:
HEX = 40, byte = 1, ciclos = 6.

Indicadores:

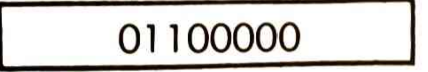


RTS **Retorno desde subrutina**

Función:

PCL ← (PILA)
S ← (S) + 1
PCH ← (PILA)
S ← (S) + 1
PC ← (PC + 1)

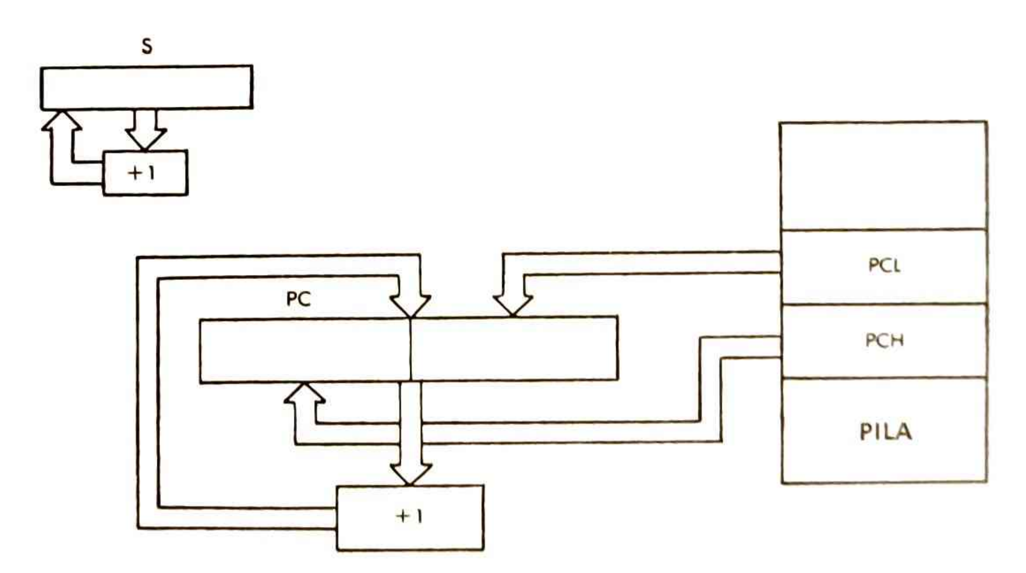
Formato:



Descripción:

Restaura el contador de programa desde la pila y lo incrementa en uno. Actualiza el puntero de pila.

Caminos de los datos:



Modo de direccionamiento:

Solamente implícito:
HEX = 60, byte = 1, ciclos = 6.

Indicadores:



SBC

Resta con acarreo

Función:

A ← (A) − DATOS − $\bar{C}$ ($\bar{C}$ es acarreo negativo)

Formato:

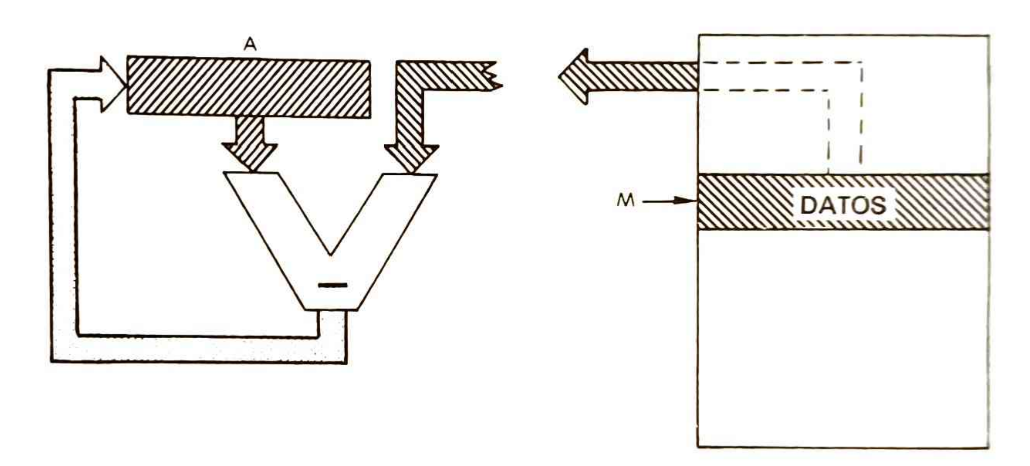
111bbb01	DIRECCIÓN/DATOS	DIRECCIÓN

Descripción:

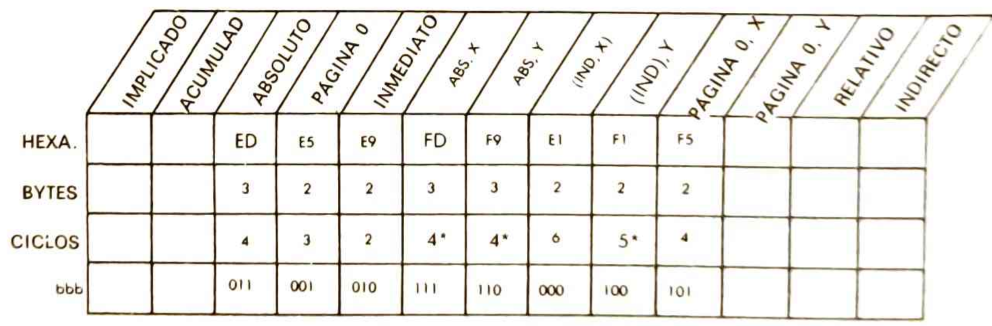
Se resta del acumulador el dato contenido en la dirección indicada, con acarreo negativo. El resultado se deja en A. Nota: SEC se utiliza para una resta sin acarreo negativo.

SBC puede ser utilizada en modo decimal o binario, dependiendo del bit D del registro de estado.

Caminos de los datos:



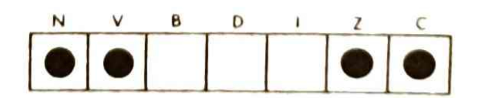
Modos de direccionamiento:



	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS. X	ABS. Y	(IND, X)	(IND), Y	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA.			ED	E5	E9	FD	F9	E1	F1	F5			
BYTES			3	2	2	3	3	2	2	2			
CICLOS			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

*: MÁS 1 CICLO SI SE CRUZA EL LIMITE DE LA PAGINA.

Indicadores:



Códigos de instrucción:

Modo	Código	Operando	bbb	HEXA.	CICLOS
ABSOLUTO	11101101	DIRECCIÓN DE 16 BITS	bbb=011	HEXA. =ED	CICLOS = 4
PÁGINA-CERO	11100101	DIRECCIÓN	bbb=001	HEXA. =E5	CICLOS = 3
INMEDIATO	11101001	DATOS	bbb=010	HEXA. =E9	CICLOS = 2
ABSOLUTO, X	11111101	DIRECCIÓN DE 16 BITS	bbb=111	HEXA. =FD	CICLOS = 4*
ABSOLUTO, Y	11111001	DIRECCIÓN DE 16 BITS	bbb=110	HEXA. =F9	CICLOS = 4*
(IND, X)	11100001	DIRECCIÓN	bbb=000	HEXA. =E1	CICLOS = 6
(IND), Y	11110001	DIRECCIÓN	bbb=100	HEXA. =F1	CICLOS = 5*
PÁGINA CERO, X	11110101	DIRECCIÓN	bbb=101	HEXA. =F5	CICLOS = 4

*: MÁS 1 CICLO SI SE CRUZA EL LÍMITE DE LA PÁGINA.

SEC

Puesta a uno del acarreo

Función:

C ← 1

Formato:

00111000

Descripción:

El bit de acarreo se pone a 1. Se utiliza antes de SBC para realizar una resta sin acarreo.

Modo de direccionamiento:

Solamente implícito:
HEX = 38, byte = 1, ciclos = 2.

Indicadores:

N	V	B	D	I	Z	C
						1

SED — Puesta en modo decimal

Función:

D ← 1

Formato:

11111000

Descripción:

El bit decimal del registro de estado se pone a 1. Cuando es 0, el modo es binario. Cuando es 1, el modo es decimal para ADC y SBC.

Modo de direccionamiento:

Solamente implícito:
HEX = F8, byte = 1, ciclos = 2.

Indicadores:

N	V	B	D	I	Z	C
			1			

SEI Puesta a uno de la inhibición de interrupción

Función:

I ← 1

Formato:

01111000

Descripción:

Se pone a 1 la máscara de interrupción. Se utiliza durante una interrupción o en un "reset" del sistema.

Modo de direccionamiento:

Solamente implícito:
HEX = 78, byte = 1, ciclos = 2.

Indicadores:

N	V	B	D	I	Z	C
				1		

STA — Almacenar en memoria el acumulador

Función:

M ← (A)

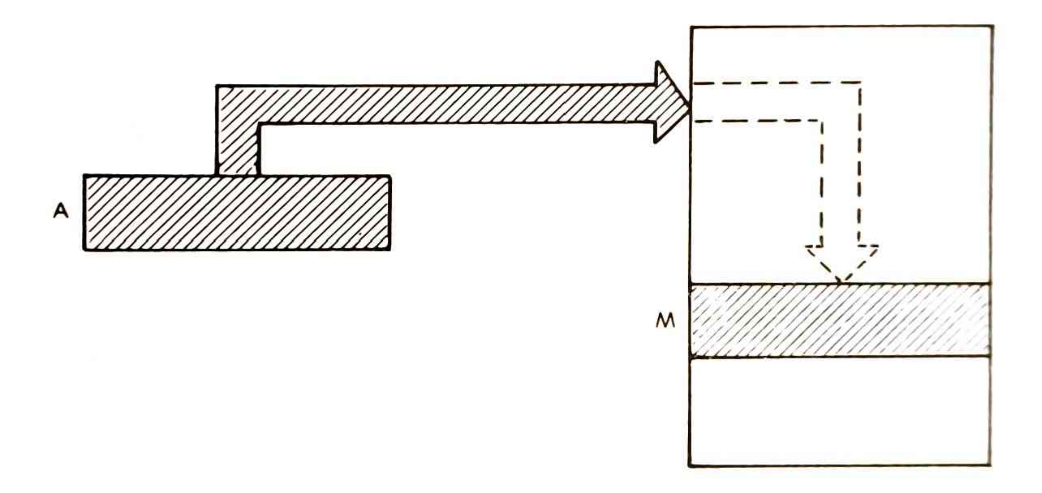
Formato:

100bbb01	DIRECCIÓN

Descripción:

El contenido de A se copia en la posición de memoria indicada. El contenido de A no cambia.

Caminos de los datos:



Modos de direccionamiento:

	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS. X	ABS. Y	(IND X)	(IND), Y	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA.			8D	85		9D	99	81	91	95			
BYTES			3	2		3	3	2	2	2			
CICLOS			4	3		5	5	6	6	4			
bbb			011	001		111	110	000	100	101			

Indicadores:



Códigos de instrucción:

Modo	Código	Operando	bbb	HEXA.	CICLOS
ABSOLUTO	10001101	DIRECCIÓN DE 16 BITS	bbb = 011	HEXA. = 8D	CICLOS = 4
PAGINA-CERO	10000101	DIRECCIÓN	bbb = 001	HEXA. = 85	CICLOS = 3
ABSOLUTO, X	10011101	DIRECCIÓN DE 16 BITS	bbb = 111	HEXA. = 9D	CICLOS = 5
ABSOLUTO, Y	10011001	DIRECCIÓN DE 16 BITS	bbb = 110	HEXA. = 99	CICLOS = 5
(IND, X)	10000001	DIRECCIÓN	bbb = 000	HEXA. = 81	CICLOS = 6
(IND), Y	10010001	DIRECCIÓN	bbb = 100	HEXA. = 91	CICLOS = 6
PÁGINA CERO, X	10010101	DIRECCIÓN	bbb = 101	HEXA. = 95	CICLOS = 4

STX Almacenar en memoria X

Función:

M ← (X)

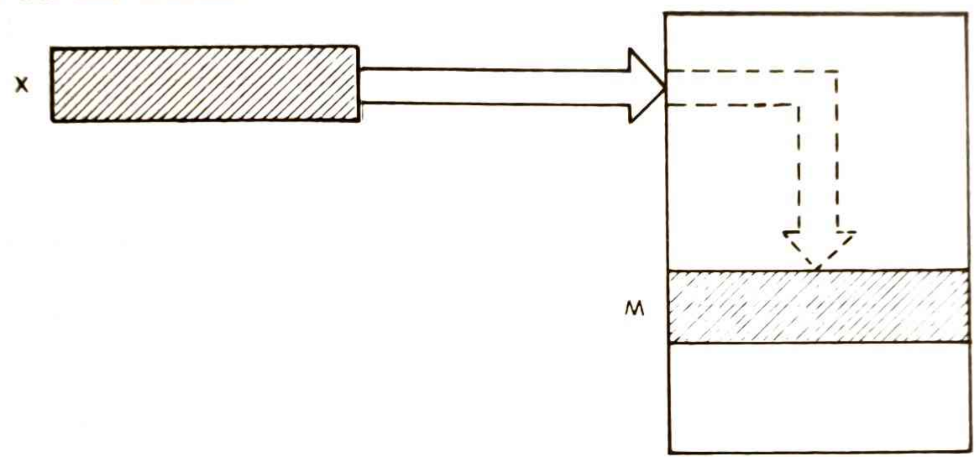
Formato:

100bb110	DIRECCIÓN

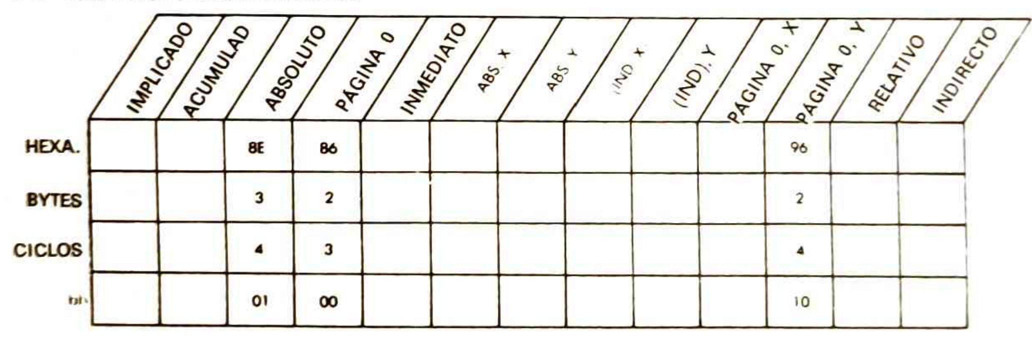
Descripción:

El contenido del registro X se copia en la posición de memoria indicada. El contenido de X se deja inalterable.

Caminos de los datos:

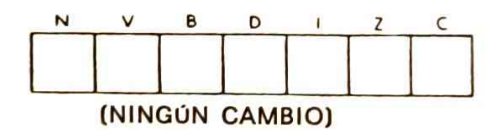


Modos de direccionamiento:



	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS. X	ABS. Y	(IND. X)	(IND), Y	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA.			8E	86							96		
BYTES			3	2							2		
CICLOS			4	3							4		
bb			01	00							10		

Indicadores:



(NINGÚN CAMBIO)

Códigos de instrucción:

ABSOLUTO: | 10001110 | DIRECCIÓN | — bb = 01 — HEXA. 8E — CICLOS = 4

PAGINA-CERO: | 10000110 | DIRECCIÓN | — bb = 00 — HEXA. 86 — CICLOS = 3

PAGINA 0, Y: | 10010110 | DIRECCIÓN | — bb = 10 — HEXA. 96 — CICLOS = 4

STY Almacenar en memoria Y

Función:

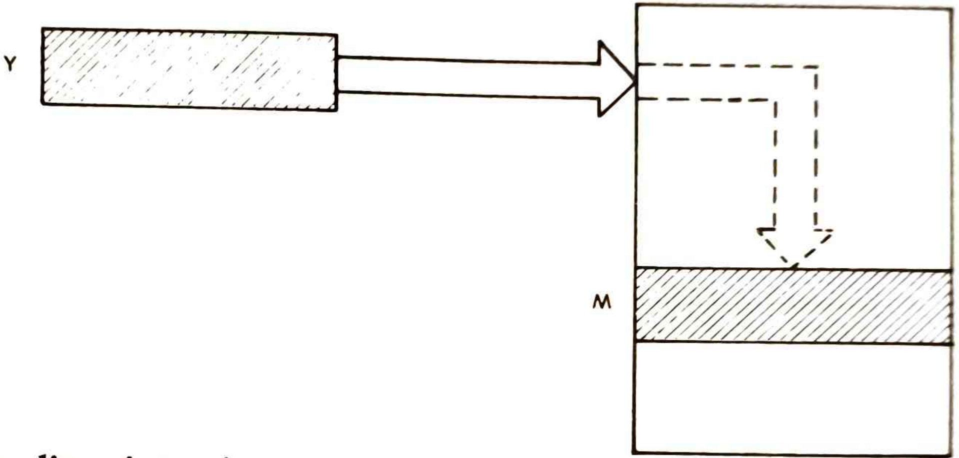
M ← (Y)

Formato: 100bb100 | DIRECCIÓN

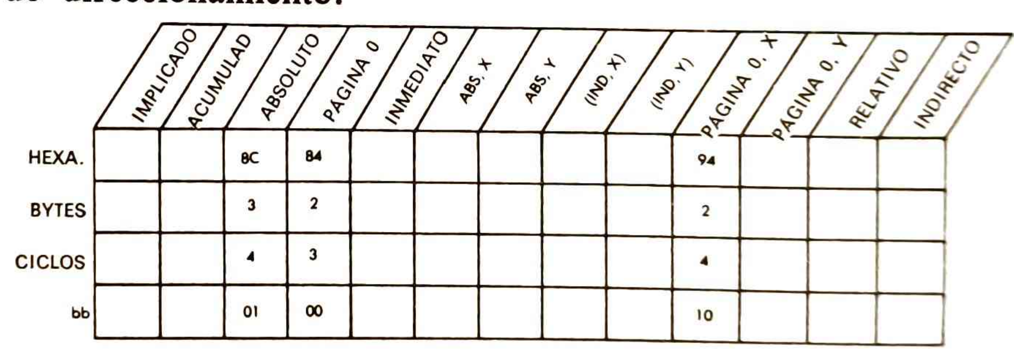
Descripción:

El contenido del registro índice Y se copia en la posición de memoria indicada. El contenido de Y se deja inalterado.

Caminos de los datos:

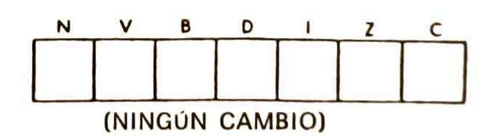


Modos de direccionamiento:

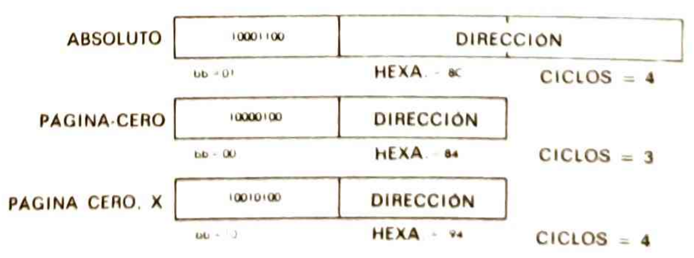


	IMPLICADO	ACUMULAD	ABSOLUTO	PAGINA 0	INMEDIATO	ABS. X	ABS. Y	(IND, X)	(IND, Y)	PAGINA 0, X	PAGINA 0, Y	RELATIVO	INDIRECTO
HEXA.			8C	84						94			
BYTES			3	2						2			
CICLOS			4	3						4			
bb			01	00						10			

Indicadores:



Códigos de instrucción:



TAX **Transferencia del acumulador a X**

Función:

X ← (A)

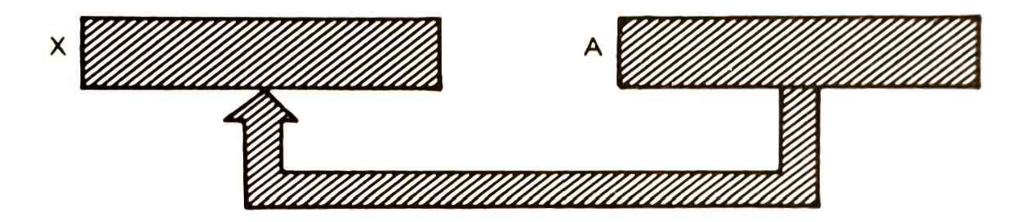
Formato:

10101010

Descripción:

El contenido del acumulador se copia al registro índice X. El contenido de A se deja inalterado.

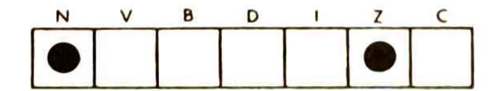
Caminos de los datos:



Modo de direccionamiento:

Solamente implícito:
HEX = AA, byte = 1, ciclos = 2.

Indicadores:



TAY   Transferencia del acumulador a Y

Función:

Y ← (A)

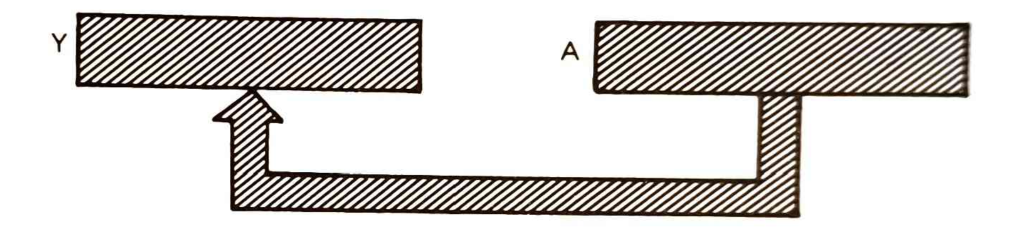
Formato:

10101000

Descripción:

Transferir el contenido del acumulador al registro índice Y. El contenido de A se deja inalterado.

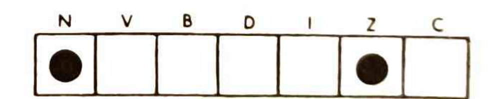
Caminos de los datos:



Modo de direccionamiento:

Solamente implícito:
HEX = A8, byte = 1, ciclos = 2.

Indicadores:



TSX Transferir S a X

Función:

X ← (S)

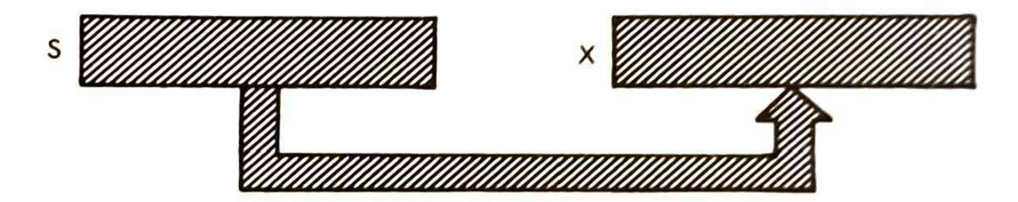
Formato:

10111010

Descripción:

El contenido del puntero de pila S se transfiere al registro índice X. El contenido de S se deja inalterable.

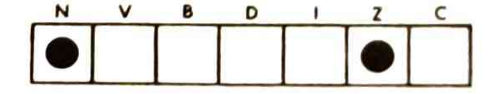
Caminos de los datos:



Modo de direccionamiento:

Solamente implícito:
HEX = BA, byte = 1, ciclos = 2.

Indicadores:



TXA **Transferir X al acumulador**

Función:

A ← (X)

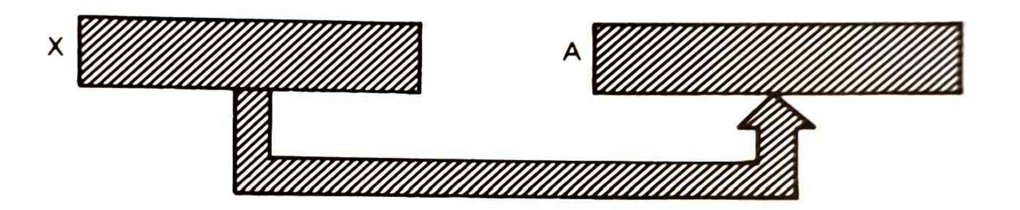
Formato:

10001010

Descripción:

El contenido del registro índice X se transfiere al acumulador. El contenido de X se deja inalterado.

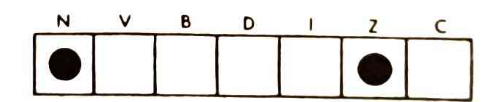
Caminos de los datos:



Modo de direccionamiento:

Solamente implícito:
HEX = 8A, byte = 1, ciclos = 2.

Indicadores:



TXS Transferir X a S

Función:

S ← (X)

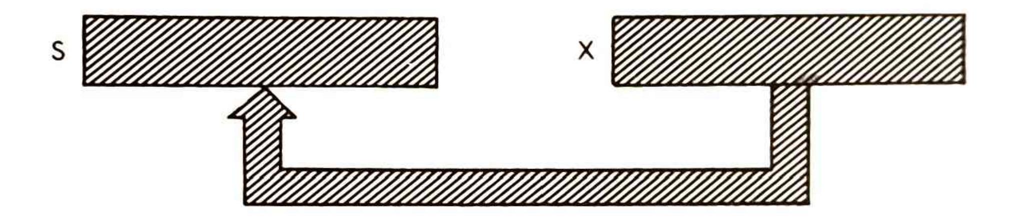
Formato:

10011010

Descripción:

El contenido del registro índice X se transfiere al puntero de pila S. El contenido de X se deja inalterado.

Caminos de los datos:

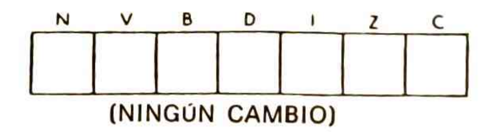


Modo de direccionamiento:

Solamente implícito:

HEX = 9A, byte = 1, ciclos = 2.

Indicadores:



TYA

Transferir Y a A

Función:

A ← (Y)

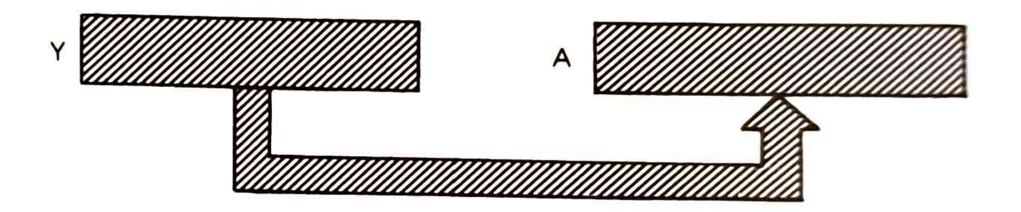
Formato:

10011000

Descripción:

El contenido del registro índice Y se transfiere al acumulador. El contenido de Y se deja inalterado.

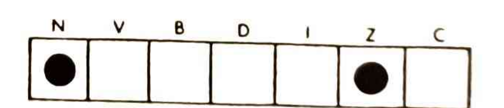
Caminos de los datos:



Modo de direccionamiento:

Solamente implícito:
HEX = 98, byte = 1, ciclos = 2.

Indicadores:



5 Técnicas de direccionamiento

INTRODUCCIÓN

En este capítulo se expondrá la teoría general del direccionamiento, así como las diversas técnicas que se han desarrollado para facilitar el acceso a los datos. En la segunda sección se pasará revista a los modos de direccionamiento específicos de los que dispone el 6502, con sus ventajas y limitaciones, si las hubiere. Finalmente, para familiarizar al lector con las diversas prestaciones posibles, una sección de aplicaciones mostrará, en el ámbito de los programas de aplicaciones concretas, las opciones posibles entre las diferentes técnicas de direccionamiento.

Habida cuenta de que el 6502 no tiene ningún registro de 16 bits que no sea el contador de programa (u ordinal) para especificar una dirección, es necesario que todo usuario del 6502 comprenda bien los diversos modos de direccionamiento y, en particular, la utilización de los registros índice. Los modos de acceso complejos, tales como una combinación de los direccionamientos indirecto e indexado, pueden omitirse en una fase inicial de su estudio. Sin embargo, todos los modos de direccionamiento son útiles para desarrollar programas eficaces para este microprocesador. Estudiemos ahora las diversas alternativas de que se dispone.

MODOS DE DIRECCIONAMIENTO

El *direccionamiento* designa la especificación, dentro de una instrucción, del emplazamiento del operando sobre el que actuará la instrucción. Los principales métodos de direccionamiento se examinarán a continuación.

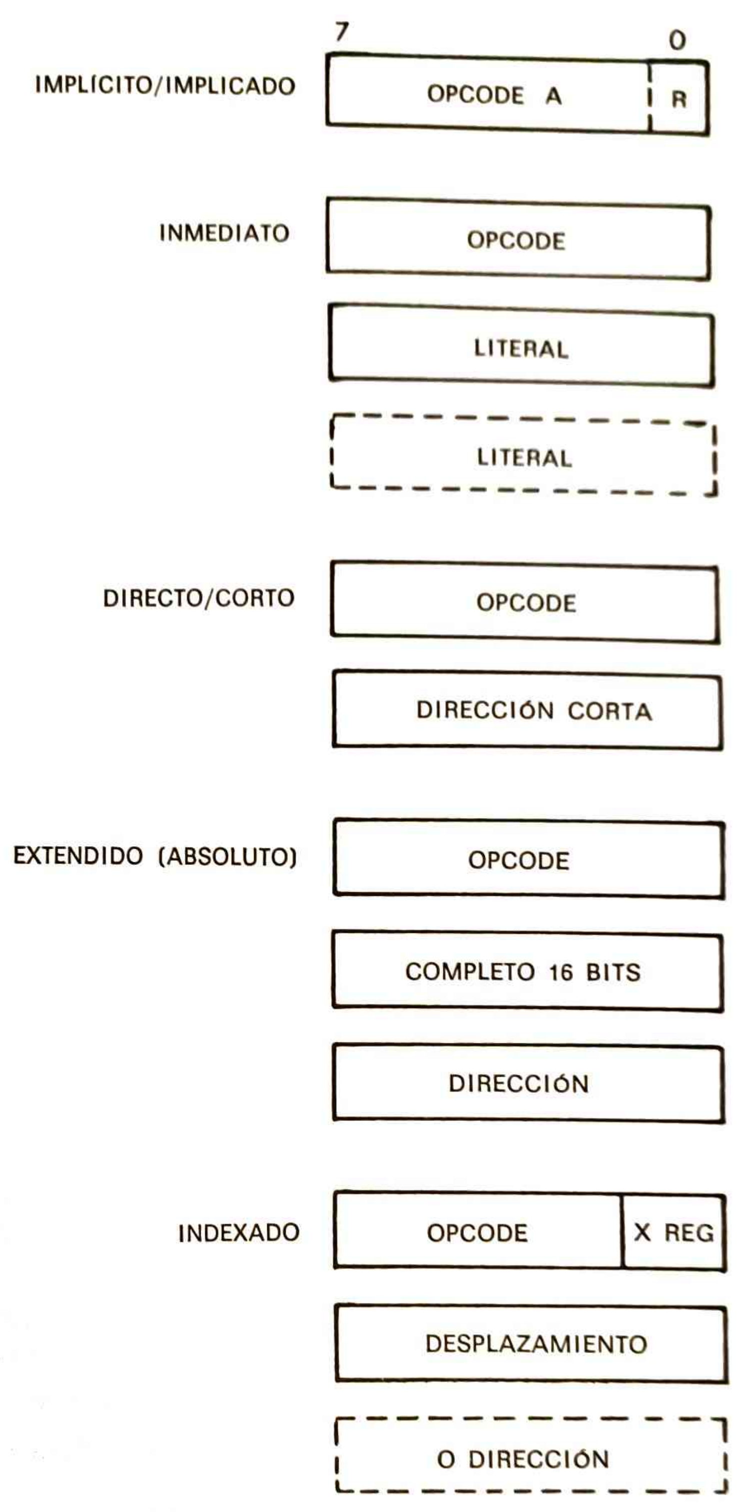


Figura 5-1 Direccionamiento.

Direccionamiento implícito

Las instrucciones que actúan exclusivamente sobre registros suelen emplear el ***direccionamiento implícito***. Ello se ilustra en la figura 5-1. Su nom-

bre se deriva del hecho de que la instrucción no contiene explícitamente la dirección del operando sobre la que actúa. En su lugar, su código de operación (OPCODE) especifica uno o varios registros, que suele ser el acumulador o cualquier otro. Puesto que los registros internos suelen ser poco numerosos (con un máximo de 8), ello necesita un pequeño número de bits. Por ejemplo, tres bits de la instrucción designarán un registro interno de entre ocho. Por consiguiente, dichas instrucciones sólo pueden codificarse en 8 bits. Esto constituye una ventaja importante, pues una instrucción de un byte (8 bits) suele ejecutarse más rápidamente que cualquier instrucción de dos o tres bytes.

Un ejemplo de una instrucción implícita para el 6502 es TAX que significa "transferir el contenido de A a X".

Direccionamiento inmediato

El direccionamiento inmediato se ilustra en la figura 5-1. El código de operación de 8 bits va seguido por un literal (una constante) de 8 o de 16 bits. Este tipo de instrucción se necesita, por ejemplo, para cargar un valor de 8 bits en un registro de 8 bits. Si el microprocesador dispone de registros de 16 bits, puede ser necesario cargar constantes de 16 bits. Ello depende de la arquitectura interna del procesador. Un ejemplo de instrucción inmediata del 6502 es ADC#0.

La segunda palabra de esta instrucción contiene el literal "0", que se añade al acumulador.

Direccionamiento absoluto

El direccionamiento absoluto designa la manera normal de tener acceso a los datos en memoria, según la cual un código de operación va seguido por una dirección de 16 bits. El direccionamiento absoluto necesita, pues, 3 bytes para la instrucción. Un ejemplo de direccionamiento absoluto es: STA $ 1234.

Esta instrucción especifica que el contenido del acumulador ha de almacenarse en la posición de memoria "1234" en hexadecimal.

El inconveniente del direccionamiento absoluto es que exige instrucciones de 3 bytes. Para mejorar la eficacia del microprocesador hay otro modo de direccionamiento a disposición del usuario, en donde una sola palabra de un byte se utiliza para la dirección: el direccionamiento directo.

Direccionamiento directo

En este modo de direccionamiento, el código de operación va seguido

por una dirección de 8 bits. Se ilustra en la figura 5-1. La ventaja de este método es que sólo necesita 2 bytes en lugar de 3 bytes como el direccionamiento absoluto. El inconveniente es que limita las direcciones a la gama 0 - 255; es decir, la página 0. Este modo se denomina también direccionamiento corto o direccionamiento de página 0. Cuando se dispone del direccionamiento corto, al direccionamiento absoluto se le suele denominar *direccionamiento extendido,* en contraste.

Direccionamiento relativo

Las instrucciones de salto normal o bifurcación requieren 8 bits para el código de operación, más los 16 bits de la dirección a la cual el programa ha de saltar. Exactamente como en el caso precedente, tiene el inconveniente de requerir tres palabras; esto es, tres ciclos de memoria. Para proporcionar bifurcaciones más eficaces, el direccionamiento relativo sólo utiliza un formato de dos palabras. La primera palabra especifica la bifurcación y suele ir acompañada de la prueba correspondiente. La segunda palabra es un desplazamiento. Como el desplazamiento debe ser positivo o negativo, una instrucción de bifurcación relativa permite una bifurcación hacia adelante de 128 posiciones (7 bits) o una bifurcación hacia atrás de 128 posiciones (más o menos 1, según los convenios). Como la mayor parte de los bucles tienden a ser cortos, las bifurcaciones relativas pueden utilizarse en la mayoría de los casos y dan lugar a rendimientos considerablemente mejorados para rutinas cortas de esta clase. Por ejemplo, ya hemos utilizado la instrucción BCC que especifica una "bifurcación si el acarreo es cero" a una posición situada a menos de 127 palabras de distancia de la instrucción de bifurcación.

Direccionamiento indexado

El direccionamiento indexado es una técnica que es útil, sobre todo para acceder sucesivamente a los elementos de un bloque o de una tabla. Ello se ilustrará más adelante mediante ejemplos en este capítulo. El principio del direccionamiento indexado es que la instrucción especifica, a la vez, un registro índice y una dirección. En el modo más general, el contenido del registro se añade a la dirección para proporcionar la dirección final. De esta manera, la dirección podría ser el comienzo de una tabla en la memoria. El registro índice se utilizaría entonces para acceder sucesivamente a todos los elementos de la tabla de forma eficaz. En la práctica suelen existir restricciones que pueden limitar la magnitud del registro índice o la magnitud del campo de desplazamiento o zona de dirección.

por una dirección de 8 bits. Se ilustra en la figura 5-1. La ventaja de este método es que sólo necesita 2 bytes en lugar de 3 bytes como el direccionamiento absoluto. El inconveniente es que limita las direcciones a la gama 0 - 255; es decir, la página 0. Este modo se denomina también direccionamiento corto o direccionamiento de página 0. Cuando se dispone del direccionamiento corto, al direccionamiento absoluto se le suele denominar *direccionamiento extendido,* en contraste.

Direccionamiento relativo

Las instrucciones de salto normal o bifurcación requieren 8 bits para el código de operación, más los 16 bits de la dirección a la cual el programa ha de saltar. Exactamente como en el caso precedente, tiene el inconveniente de requerir tres palabras; esto es, tres ciclos de memoria. Para proporcionar bifurcaciones más eficaces, el direccionamiento relativo sólo utiliza un formato de dos palabras. La primera palabra especifica la bifurcación y suele ir acompañada de la prueba correspondiente. La segunda palabra es un desplazamiento. Como el desplazamiento debe ser positivo o negativo, una instrucción de bifurcación relativa permite una bifurcación hacia adelante de 128 posiciones (7 bits) o una bifurcación hacia atrás de 128 posiciones (más o menos 1, según los convenios). Como la mayor parte de los bucles tienden a ser cortos, las bifurcaciones relativas pueden utilizarse en la mayoría de los casos y dan lugar a rendimientos considerablemente mejorados para rutinas cortas de esta clase. Por ejemplo, ya hemos utilizado la instrucción BCC que especifica una "bifurcación si el acarreo es cero" a una posición situada a menos de 127 palabras de distancia de la instrucción de bifurcación.

Direccionamiento indexado

El direccionamiento indexado es una técnica que es útil, sobre todo para acceder sucesivamente a los elementos de un bloque o de una tabla. Ello se ilustrará más adelante mediante ejemplos en este capítulo. El principio del direccionamiento indexado es que la instrucción especifica, a la vez, un registro índice y una dirección. En el modo más general, el contenido del registro se añade a la dirección para proporcionar la dirección final. De esta manera, la dirección podría ser el comienzo de una tabla en la memoria. El registro índice se utilizaría entonces para acceder sucesivamente a todos los elementos de la tabla de forma eficaz. En la práctica suelen existir restricciones que pueden limitar la magnitud del registro índice o la magnitud del campo de desplazamiento o zona de dirección.

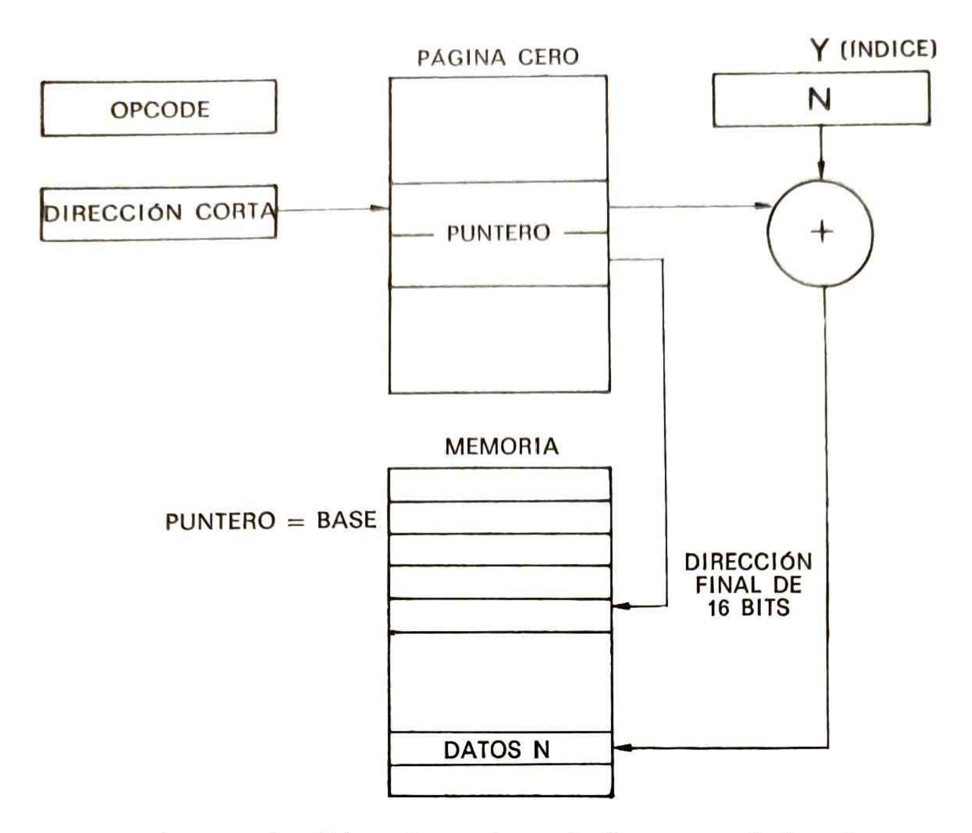


Figura 5-2 Direccionamiento indirecto postindexado.

Preindexación y postindexación

Pueden distinguirse dos modos de indexación. La preindexación es el modo de indexación habitual en donde la dirección final es la suma de una dirección o de un desplazamiento y del contenido del registro índice.

La postindexación considera el contenido del campo de desplazamiento como la *dirección* del desplazamiento real y no como el desplazamiento en sí mismo. Esto se ilustra en la figura 5-2. En direccionamiento postindexado, la dirección final es la suma del contenido del registro índice más el contenido de la palabra de memoria, *designada por el campo o zona de desplazamiento.* Esto utiliza, de hecho, una combinación de la preindexación y del direccionamiento indirecto. Pero todavía no hemos definido el direccionamiento indirecto, por lo que vamos a hacerlo a continuación.

Direccionamiento indirecto

Ya hemos visto el caso en que dos subrutinas (subprogramas) pueden desear intercambiar una gran cantidad de datos almacenados en memoria. Con un carácter más general, varios programas o varias subrutinas pueden tener necesidad de acceder a un bloque común de información. Para con-

servar la generalidad del programa, es deseable no guardar dicho bloque en una dirección de memoria fija. En particular, la magnitud de este bloque puede aumentar o disminuir de manera dinámica y puede tener que residir en diferentes zonas de la memoria, en función de su magnitud. No sería, pues, factible pretender acceder a este bloque con direcciones absolutas.

La solución de este problema radica en depositar la dirección de comienzo del bloque en una posición de memoria fija. Esto es análogo a la situación en que varias personas tienen necesidad de entrar en una casa y sólo se dispone de una llave. Por convenio, la llave de entrada a la casa se esconderá bajo el felpudo. Cada usuario sabrá, pues, en donde mirar (bajo el felpudo) para encontrar la llave de la casa (o, quizás, para encontrar la dirección de una reunión prevista, para establecer una analogía más exacta). El direccionamiento indirecto pone, pues, en juego un código de operación de 8 bits seguido por una dirección de 16 bits. Esta dirección se utiliza simplemente para recuperar una palabra de memoria. Normalmente, esta última será una palabra de 16 bits (en nuestro caso, dos bytes) en la memoria, lo cual se ilustra en la figura 5-3. Los dos bytes en la dirección especificada, A1, contienen A2. Y esta última se interpreta entonces como la dirección efectiva, o real, de los datos que se desea manipular.

El direccionamiento indirecto es particularmente útil cada vez que se utilizan punteros. Diversas zonas del programa pueden referirse a estos punteros para acceder a una palabra o a un bloque de datos de una forma cómoda y elegante.

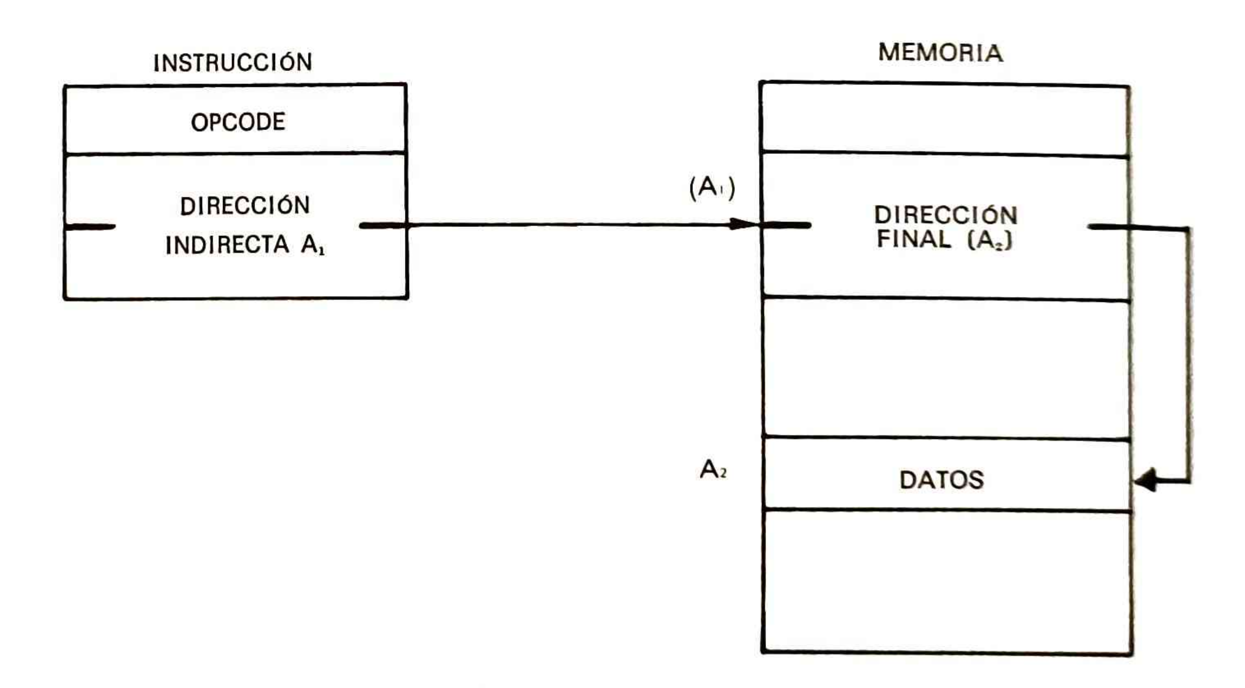


Figura 5-3 Direccionamiento indirecto.

Combinación de modos

Los anteriores modos de direccionamiento pueden combinarse. En particular, debe ser posible en un esquema de direccionamiento completamente general para utilizar muchos niveles de "indirección" (direccionamiento indirecto). La dirección A2 podría interpretarse, de nuevo, como una dirección indirecta y así sucesivamente.

El direccionamiento indexado puede combinarse también con acceso indirecto. Ello permite acceder eficazmente a la enésima palabra de un bloque de datos, con tal de que se sepa en donde se encuentra el puntero hacia la dirección de comienzo.

Ahora ya estamos familiarizados con todos los modos de direccionamiento habituales que pueden proporcionarse en un sistema. La mayor parte de los sistemas de microprocesadores, debido a la limitación de la complejidad de una MPU, que debe realizarse en una sola pastilla, no proporciona todos los modos posibles sino solamente un subconjunto limitado de estos modos. El 6502 proporciona un conjunto excepcionalmente amplio de posibilidades. Vamos a examinarlas a continuación.

MODOS DE DIRECCIONAMIENTO DEL 6502

Direccionamiento implícito (6502)

El direccionamiento implícito se utiliza mediante una instrucción de un solo byte que actúa sobre los registros internos. Cuando las instrucciones implícitas actúan exclusivamente sobre los registros internos, su ejecución sólo requiere dos ciclos. Cuando acceden a la memoria, requieren tres ciclos.

Las instrucciones *que actúan exclusivamente en el interior del 6502,* son: CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS y TYA.

Las instrucciones que requieren acceso de memoria son: BRK, PHA, PHP, PLA, PLP, RTI y RTS.

Estas instrucciones se han descrito en el capítulo anterior y su modo de funcionamiento debe estar claro.

Direccionamiento inmediato (6502)

Como el 6502 sólo tiene registros de trabajo de 8 bits (PC no es un registro de trabajo), el direccionamiento inmediato está limitado, en el caso del 6502, a constantes de 8 bits. Todas las instrucciones tienen, por consiguiente, en el modo de direccionamiento inmediato, dos bytes de longitud.

El primer byte contiene el código de operación y el segundo byte contiene la constante, o literal, que ha de cargarse en un registro o combinarse con uno de los registros en una operación aritmética o lógica.

Las instrucciones que utilizan este modo son: ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA y SBC.

Direccionamiento absoluto (6502)

Por definición, el direccionamiento absoluto requiere tres bytes. El primero es el código de operación y los dos bytes siguientes forman la dirección de 16 bits que especifica la posición del operando. Salvo en el caso de un salto absoluto, este modo de direccionamiento requiere cuatro ciclos.

Las instrucciones que pueden utilizar el direccionamiento absoluto son: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, JMP, JSR, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX y STY.

Direccionamiento de página cero (6502)

Por definición, el direccionamiento de página cero ocupa dos bytes: el primero es para el código de operación y el segundo es para la dirección corta o de 8 bits.

El direccionamiento de página cero requiere tres ciclos. Como este direccionamiento ofrece una ventaja importante en velocidad y en ocupación de memoria (código más corto), debe utilizarse siempre que sea posible. Ello exige una gestión cuidadosa de la memoria por parte del programador. En términos generales, se pueden considerar las 256 primeras palabras de la memoria como el conjunto de los registros de trabajo del 6502. Cualquier instrucción actúa sobre estos 256 "registros" en solamente tres ciclos. Este espacio debe, pues, reservarse cuidadosamente para los datos esenciales a los que es preciso acceder muy rápidamente.

Las instrucciones que pueden utilizar el direccionamiento de página cero son las que emplean el direccionamiento absoluto, con la excepción de JMP y JSR (que exigen una dirección de 16 bits).

La lista de las instrucciones "legales" es: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX y STY.

Direccionamiento relativo (6502)

Por definición, el direccionamiento relativo utiliza dos bytes. El primero es la instrucción de salto, mientras que el segundo especifica el desplazamiento y su signo. Con el fin de diferenciar este modo respecto a la instrucción de

salto absoluto, aquí se les denomina *bifurcaciones*. En el caso del 6502, las bifurcaciones utilizan siempre el direccionamiento relativo. Los saltos emplean siempre el direccionamiento absoluto (más, naturalmente, los demás submodos que pueden combinarse con los mismos, como los direccionamientos indirecto e indexado).

Desde el punto de vista del tiempo de ejecución, estas instrucciones deben examinarse con precaución. Cuando una prueba es negativa, es decir, cuando no hay bifurcación, la instrucción requiere dos ciclos solamente. Ello se debe a que el contador de programa apunta ya hacia la próxima instrucción a ejecutar. Pero, cuando la prueba es positiva, es decir, cuando la bifurcación debe tener lugar, la instrucción requiere tres ciclos y es preciso calcular una nueva dirección efectiva. La actualización del contador de programa requiere un ciclo suplementario. Sin embargo, si la bifurcación franquea una frontera de página, se necesita una actualización suplementaria del contador de programa y la duración efectiva de la instrucción se hace de cuatro ciclos.

Desde el punto de vista lógico, el usuario no necesita preocuparse del cruce de una frontera de página, pues el hardware se ocupa de ello. Pero como hay un acarreo positivo, o negativo, suplementario en cada cruce de frontera de página, se cambiará el tiempo de ejecución de la bifurcación. Si esta bifurcación forma parte de un bucle, cuyo tiempo es crítico, es preciso prestarle atención.

Normalmente, en el momento en que se ensambla el programa, un buen ensamblador indica al programador que una bifurcación franquea una frontera de página, para el caso en que el tiempo de ejecución fuera crítico.

Cuando no se tenga seguridad de que la bifurcación vaya a tener lugar, se debe considerar que, en ciertos casos, la instrucción requerirá dos ciclos y, en otros casos, tres. Se suele utilizar un tiempo medio.

Las únicas instrucciones que permiten el direccionamiento relativo en el 6502 son las bifurcaciones. Hay 8 instrucciones de bifurcación que prueban los indicadores del registro de estado para el valor "0" o "1". Éstas son: BCC, BCS, BEQ, BMI, BNE, BPL, BVC y BVS.

Direccionamiento indexado (6502)

El 6502 no dispone de un direccionamiento indexado completamente general, sino solamente uno limitado. Tiene dos registros índice. Pero estos registros están limitados a 8 bits. El contenido de un registro índice se añade a la parte de dirección de la instrucción. El registro índice suele utilizarse como contador para acceder sucesivamente a los elementos de un bloque o de una tabla. Esta es la razón por la que existen instrucciones especiales para incrementar, o decrementar, cada uno de los registros índice por separado. Además, hay dos instrucciones especiales para comparar el contenido de

los registros índice con respecto a una posición de memoria, que constituyen un medio importante para el empleo efectivo de los registros índice para las pruebas con respecto a un límite.

En la práctica, la mayor parte de las tablas utilizadas suelen ser más cortas que 256 elementos y la limitación de los registros índice a 8 bits no suele tener consecuencias desfavorables. El direccionamiento indexado puede utilizarse no solamente con direcciones absolutas ordinarias, es decir, zonas de dirección de 16 bits, sino también con direcciones de página cero, esto es, zonas de direcciones de 8 bits. No hay más que una restricción. El registro X puede utilizarse en los dos casos. Pero el registro Y sólo permite el direccionamiento *absoluto* indexado y no el de *página cero indexado* (salvo para LDX y STX que pueden modificarse por Y).

El direccionamiento absoluto indexado requiere cuatro ciclos, a no ser que se cruce una frontera de página, en cuyo caso son precisos cinco ciclos.

Las instrucciones absolutas indexadas pueden utilizar registros X o Y para proporcionar el campo de desplazamiento. La lista de las instrucciones que pueden emplear este modo es:

— Con X: ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC y STA (no STY);
— con Y: ADC, AND, CMP, EOR, LDA, LDX, ORA, SBC y STA (pero sin ASL, DEC, LSR, ROL y ROR).

En lo que respecta al direccionamiento de página cero indexado, el registro X es el único autorizado (salvo para LDX y STX). Las instrucciones permitidas son: ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA y STY.

Direccionamiento indirecto (6502)

El 6502 no tiene un direccionamiento indirecto completamente general. Restringe la zona de dirección a 8 bits. Dicho de otro modo, todas las direcciones indirectas utilizan el submodo, de "página cero". La dirección efectiva sobre la que va a actuar el código de operación está constituida, pues, por 16 bits especificados por la dirección de página cero situada en la instrucción. Además, no puede tener lugar ninguna "indirección" suplementaria. Ello significa que la dirección encontrada en la página cero debe utilizarse "tal cual" y no puede emplearse para una nueva.

Finalmente, todos los accesos indirectos deben indexarse, salvo para JMP.

Sin embargo, es preciso destacar que muy pocos microprocesadores proporcionan direccionamiento indirecto, cualquiera que sea. Finalmente, siem-

pre es posible implantar un direccionamiento indirecto más general con la ayuda de una macrodefinición.

Dos modos de direccionamiento indirecto son posibles: direccionamiento indirecto (pre) indexado y direccionamiento (post) indexado indirecto (con la excepción de JMP que utiliza indirecto puro).

Direccionamiento indirecto indexado

En este modo se añade el contenido del registro índice X a la dirección de página cero para obtener la dirección final de 16 bits. Se trata de una manera eficaz de recuperar uno de varios datos posibles, hacia los cuales apuntan unos punteros cuyo número está contenido en el registro índice X. Esto se ilustra en la figura 5-4.

En esta ilustración, la página cero contiene una tabla de punteros. El primer puntero está en la dirección A, que forma parte de la instrucción. Si el contenido de X es 2N, entonces esta instrucción accederá al puntero número N de la tabla y recuperará los datos hacia los cuales apunta.

El direccionamiento indirecto indexado requiere 6 ciclos. Es naturalmente más lento que cualquier modo de direccionamiento directo. Su ventaja es la flexibilidad de programación o la mejora de la velocidad total que proporciona.

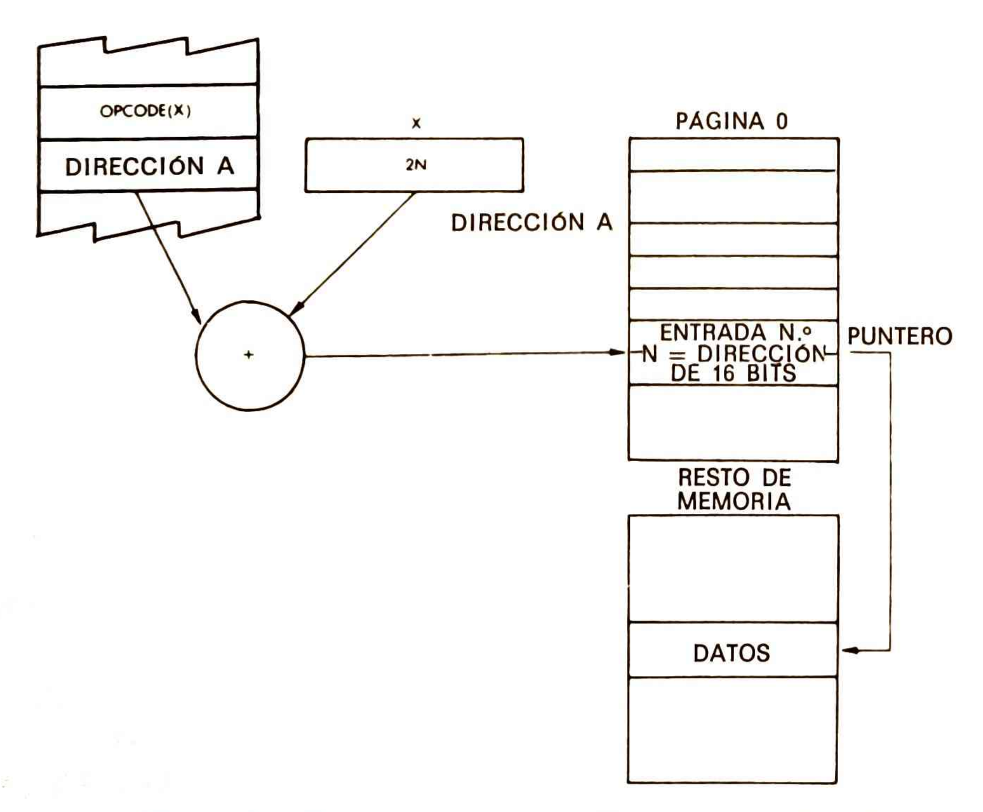


***Figura 5-4** Direccionamiento indirecto (pre) indexado.*

Las instrucciones permitidas son: ADC, AND, CMP, EOR, LDA, ORA, SBC y STA.

Direccionamiento indexado indirecto

Corresponde al mecanismo de postindexación que se ha descrito en la sección anterior. Aquí, la indexación se realiza después de la "indirección" y no antes. Dicho de otro modo, la dirección corta, que figura en la instrucción, sirve para acceder a un puntero de 16 bits en página cero. El contenido del registro índice Y se añade, entonces, como un desplazamiento a este puntero. A continuación, se recuperan los datos finales (fig. 5-2).

En este caso, el puntero contenido en página cero indica la base de una tabla en memoria. El registro índice Y proporciona un desplazamiento. Forma un verdadero índice en el interior de la tabla. Esta instrucción es particularmente potente para acceder al enésimo elemento de una tabla, con tal de que la dirección de comienzo de la tabla se conserve en página cero. Puede hacerse con dos bytes solamente.

Las instrucciones permitidas ("legales") son: ADC, AND, CMP, EOR, LDA, ORA, SBC y STA.

Excepción: Instrucción de salto

La instrucción de salto puede utilizar direccionamiento indirecto absoluto. Es la única instrucción que puede emplear este modo.

UTILIZACIÓN DE LOS MODOS DE DIRECCIONAMIENTO DEL 6502

Direccionamientos largo y corto

Ya hemos utilizado instrucciones de bifurcación en los diferentes programas que hemos desarrollado. Son autoexplicatorias. Una cuestión interesante que se plantea es: ¿qué podemos hacer si el margen admisible para la bifurcación no es suficiente para nuestras necesidades? Una solución sencilla es utilizar lo que se denomina *bifurcación larga*. Se trata simplemente de una bifurcación a una posición que contiene una instrucción de salto:

```
BCC + 3                     BIFURCACIÓN A DIRECCIÓN CO-
                            RRIENTE + 3 SI C = 0
JMP LEJOS                   DE NO SER ASÍ, SALTAR A LEJOS
(INSTRUCCIÓN SIGUIENTE)
```

El anterior programa de dos líneas dará lugar a la bifurcación a la posición "LEJOS", si el acarreo está en "1". Esto resuelve nuestro problema de bifurcación larga o a distancia. Consideremos, ahora, los modos de direccionamiento más complejos tales como la indexación y la "indirección".

Utilización del direccionamiento indexado para accesos secuenciales en un bloque

La indexación se utiliza principalmente para direccionar posiciones sucesivas dentro de una tabla. La restricción es que el desplazamiento máximo debe ser inferior a 256, con el fin de que pueda residir en un registro índice de 8 bits.

Hemos aprendido cómo reconocer el carácter "*". Ahora vamos a probar en una tabla de 100 elementos la presencia de un carácter "*". La dirección de partida de esta tabla se denomina BASE. La tabla sólo tiene 100 elementos. Este número es inferior a 256 y se puede, pues, utilizar un registro índice. El programa aparece como se indica a continuación:

```
BÚSQUEDA                   LDX #0
SIGUIENTE                  LDA BASE, X
                           CMP #'*
                           BEQ ASTERISCO ENCONTRADO
                           INX
                           CPX #100
                           BNE SIGUIENTE
NO ENCONTRADO              ...
ASTERISCO ENCONTRADO       ...
```

El diagrama de flujo correspondiente a este programa se ilustra en la figura 5-5. Se recomienda verificar que el programa le es realmente equivalente. La lógica del programa es sencilla. El registro X se utiliza para apuntar hacia el elemento considerado de la tabla. La segunda instrucción del programa:

SIGUIENTE LDA BASE, X

utiliza direccionamiento indexado absoluto. Especifica que el acumulador ha de cargarse a partir de la dirección BASE (dirección absoluta de 16 bits) más el contenido de X. Al principio, el contenido de X es 0. El primer elemento al que se accede es el que se encuentra en la dirección BASE. Puede verse que después de la siguiente iteración, X tendrá el valor "1" y se accederá al siguiente elemento secuencial de la tabla, en la dirección BASE + 1.

La tercera instrucción del programa, CMP #'* compara el valor del

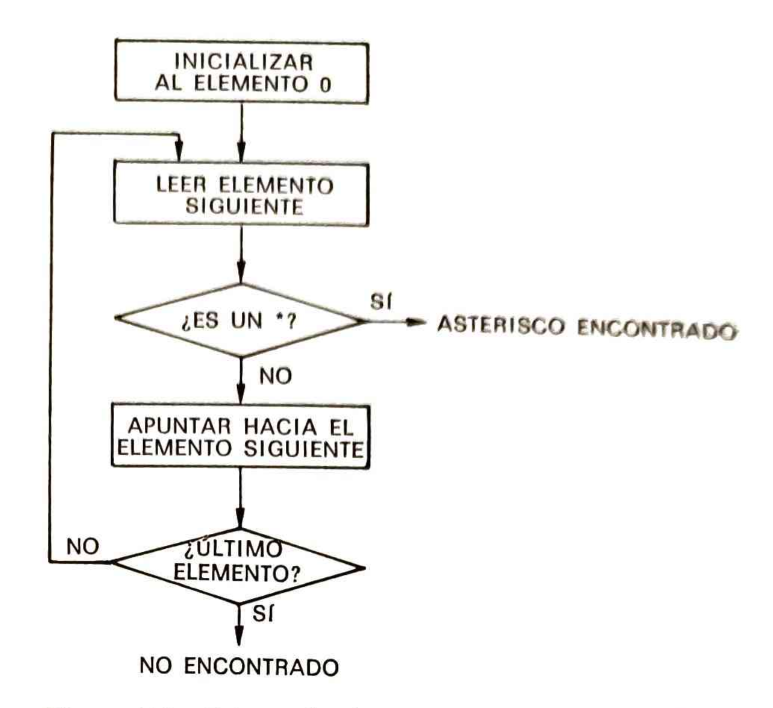


Figura 5-5 Búsqueda de un carácter en una tabla.

carácter que se ha leído en el acumulador con el código para "*". La siguiente instrucción prueba los resultados de la comparación. Si se ha encontrado una coincidencia, la bifurcación se produce a la etiqueta ASTERISCO ENCONTRADO:

BEQ ASTERISCO ENCONTRADO

De no ser así, se ejecuta la instrucción secuencial siguiente:

INX

El contador de índice se aumenta en 1. Al hacer referencia al diagrama de flujo de la figura 5-5, en su parte inferior, se constata que el valor actual del registro de índice debe probarse para conseguir que no se salga de los límites de la tabla (en este caso, 100 elementos). Esto se realiza mediante la instrucción siguiente:

CPX #100

Esta instrucción compara el registro X con el valor $100. Si la prueba es negativa, debemos continuar buscando el carácter siguiente. Esto se realiza mediante:

BNE SIGUIENTE

Esta instrucción especifica una bifurcación a la etiqueta SIGUIENTE (segunda instrucción del programa), si no hay igualdad. El bucle se ejecutará en tanto que no se haya encontrado un "*" o mientras no se alcance el valor "100" en el índice. Entonces se ejecutará la siguiente instrucción secuencial "NO ENCONTRADO". Corresponde al caso en que no se haya encontrado un "*".

Las operaciones ejecutadas según que se haya encontrado, o no, un "*" no tienen importancia en este caso y se especificarían por el programador.

Hemos aprendido a utilizar el modo de direccionamiento indexado para acceder a elementos sucesivos en una tabla. Utilicemos esta nueva aptitud y aumentemos un poco la dificultad. Vamos a desarrollar un programa utilitario importante capaz de copiar un bloque de una zona de memoria en otra. Supondremos inicialmente que el número de elementos del bloque es inferior a 256, de modo que podamos utilizar el registro índice X. A continuación, consideraremos el caso general en que sea superior a 256.

Rutina de transferencia de bloque de menos de 256 elementos

Llamaremos "NÚMERO" al número de elementos del bloque a transferir. El número se supone que es inferior a 256. BASE es la dirección de comienzo del bloque. DEST es la dirección de base de la zona de memoria de destino. El algoritmo es muy sencillo: se transfiere una palabra cada vez y se conserva el seguimiento de la palabra que ha de transferirse almacenando su posición en el registro índice X. El programa aparece en la forma siguiente:

```
                LDX   #NÚMERO
SIGUIENTE       LDA   BASE X
                STA   DEST X
                DEX
                BNE   SIGUIENTE
```

Examinémoslo:

LDX # NÚMERO

Esta línea del programa carga el número N de palabras a transferir en el registro índice. La instrucción siguiente carga la palabra #N del bloque en el acumulador y la tercera la deposita en la zona de destino (fig. 5-6).

OBSERVACIÓN: Este programa sólo funcionará correctamente si la dirección BASE se supone puntero *por debajo* del bloque, del mismo modo que para la dirección DEST. De no ser así, es preciso un ajuste del programa.

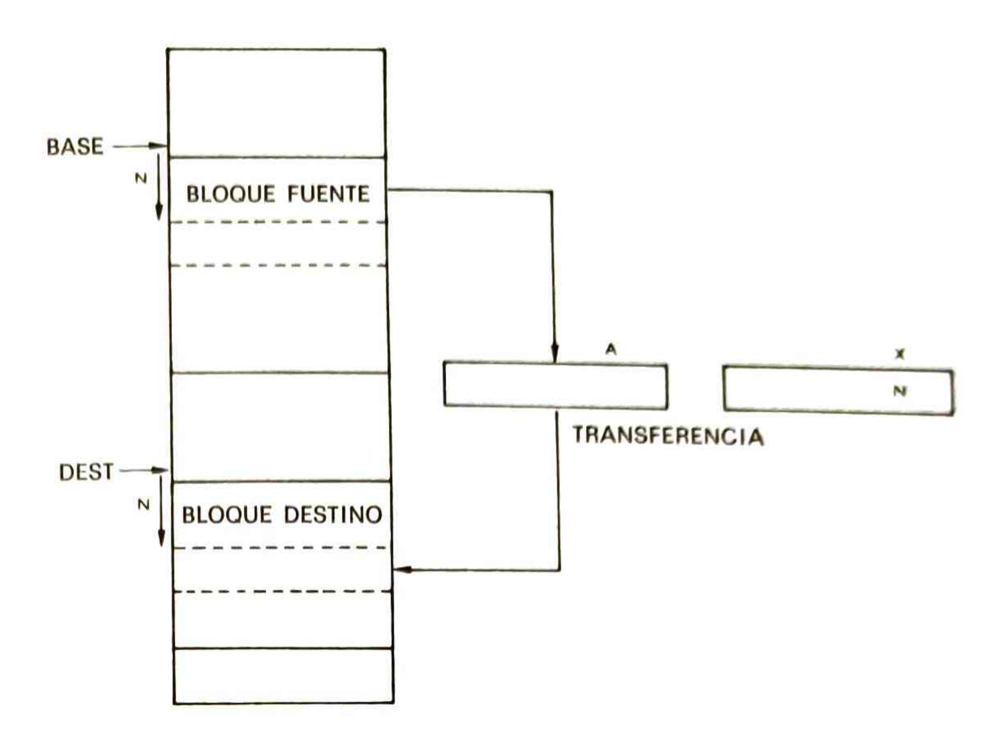


Figura 5-6 Organización de la memoria para la transferencia de bloques.

Después de que se haya transferido una palabra desde el origen a la zona de destino, debe actualizarse el registro índice. Esto se realiza mediante la instrucción DEX, que lo decrementa. A continuación, el programa prueba simplemente si X ha decrementado a "0". Si es así, se termina el programa. De no ser así, se realiza un bucle volviendo a la etiqueta "SIGUIENTE".

Destacaremos que cuando X = 0, el programa *no realiza un bucle*. Por consiguiente, no se transferirá la palabra situada en la dirección BASE. La última palabra transferida es la de la dirección BASE + 1. Esta es la razón por la que hemos supuesto que la base apuntaba justamente *por debajo* del bloque.

Ejercicio 5.1: *Modificar el programa anterior, suponiendo que BASE y DEST apuntaban al primer elemento del bloque.*

Este programa ilustra también la utilización de los contadores de bucle. Se observará que X se ha cargado con el *valor final*, luego se *decrementa* y se prueba. A primera vista podría parecer más sencillo comenzar con "0" en X y luego incrementarlo hasta que alcance el valor máximo. Pero para comprobar si X ha alcanzado su máximo sería preciso una instrucción suplementaria (la instrucción de comparación). El bucle requeriría entonces cinco instrucciones en lugar de cuatro. Como este programa de transferencia se utilizará normalmente para grandes números de palabras, es importante re-

ducir el número de instrucciones del bucle. Esta es la razón por la que, al menos para bucles cortos, el registro índice es preferiblemente *decrementado* y no *incrementado*.

Rutina de transferencia de bloque (más de 256 elementos)

Consideremos el caso general de la transferencia de un bloque que puede contener más de 256 elementos. Ya no podemos utilizar un registro índice único pues 8 bits no bastan para almacenar un número mayor que 256. La organización de la memoria relativa a este programa se ilustra en la figura 5-7. La longitud del bloque de memoria a transferir requiere 16 bits y, por consiguiente, se almacena en memoria. La parte alta representa el número de bloques de 256 palabras: "BLOQUES". La parte restante se denomina "RESTO" y representa el número de palabras a transferir cuando todos los bloques de 256 palabras hayan sido objeto de transferencia. Las direcciones de fuente y de destino estarán en memoria en las posiciones DESDE y A. Supongamos,

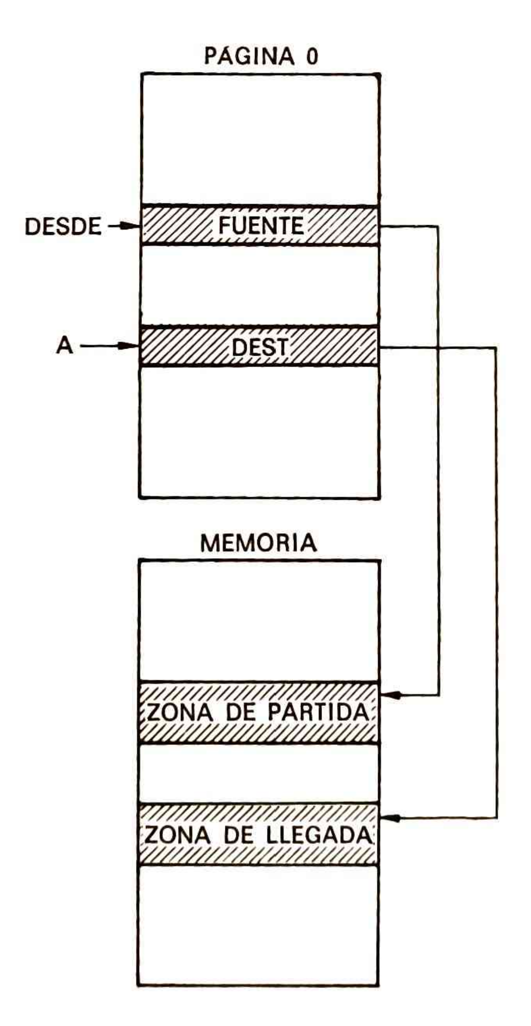


Figura 5-7 Mapa de memoria para la transferencia de bloque general.

primero, que RESTO es nulo, es decir, que se transfiere un número de bloques de 256 palabras. El programa correspondiente será como sigue:

```
            LDA   #FUENTE BAJA
            STA   DESDE
            LDA   #FUENTE ALTA
            STA   DESDE+1        ALMACENAR DIRECCIÓN
                                  DE PARTIDA
            LDA   #DEST BAJO
            STA   A
            LDA   #DEST ALTO
            STA   A+1            ALMACENAR DIRECCIÓN
                                  DE LLEGADA
            LDX   #BLOQUES       NÚMERO DE BLOQUES
            LDY   #0             TAMAÑO DEL BLOQUE
SIGUIENTE   LDA   (DESDE), Y     LEER UN ELEMENTO
            STA   (A), Y         TRANSFERIBLE
            DEY                  PUNTERO DE PALABRA
                                  ACTUALIZADO
            BNE   SIGUIENTE      ¿ACABADO?
BLOQUE SIG. INC   DESDE+1        INCREMENTAR PUNTERO
                                  DE BLOQUE
            INC   A+1            IGUAL
            DEX                  CONTADOR DE BLOQUES
            BMI   FIN
            BNE   SIGUIENTE
            LDY   #RESTO
            BNE   SIGUIENTE
```

La dirección de fuente de 16 bits se almacena por las primeras cuatro instrucciones en la dirección de memoria "DESDE". Las siguientes cuatro instrucciones hacen lo mismo para la dirección de destino, que se almacena en "A". Puesto que tenemos que transferir un número de palabras superior a 256, utilizaremos simplemente dos registros índice de 8 bits. La siguiente instrucción carga el registro X con el número de bloques a transferir. Esta es la instrucción 9.ª en el programa. La siguiente instrucción carga el valor cero en el registro de índice Y para inicializarlo para la transferencia de 256 palabras. Ahora utilizaremos direccionamiento indirecto indexado. Debe recordarse que este último dará lugar, primero, a una "indirección" dentro de la página cero y, luego, a un acceso indexado a la dirección de 16 bits especificada por el registro índice. Consideremos el programa:

```
SIGUIENTE LDA (DESDE), Y
```

Esta instrucción carga el acumulador con el contenido de la posición de memoria, cuya dirección es la de fuente, más el contenido del registro índice Y. Examinemos la figura 5-7 para el mapa de implantación de memoria. En este caso, el contenido del registro Y es inicialmente 0. Por consiguiente, "A" se cargará con el contenido de la dirección de memoria "FUENTE". Obsérvese que en este caso, a diferencia del ejemplo anterior, "FUENTE" es la dirección de la primera palabra dentro del bloque.

Con el empleo de la misma técnica, la siguiente instrucción depositará el contenido del acumulador (la primera palabra del bloque que deseamos transferir) en la posición de destino adecuada:

STA (A), Y

Exactamente como el caso anterior, decrementamos simplemente el registro índice y luego efectuamos un bucle 256 veces. Esto se realiza por las instrucciones siguientes:

DEY
BNE SIGUIENTE

Observación: Aquí se utiliza un artificio de programación para disminuir la magnitud del programa. El lector atento se habrá percatado de que el registro índice Y está *decrementado*. La primera palabra a transferir será, pues, la palabra en la posición 0. La siguiente será la palabra n.º 255. Ello se debe al hecho de que, si se decrementa "0", se obtienen ocho "1" en el registro (o sea 255). El lector se cerciorará de que no hay error. Cuando el registro Y se decrementa para llegar a "0", *no habrá* transferencia. La siguiente instrucción ejecutada será BLOQUE SIG. En consecuencia, se habrán transferido exactamente 256 palabras. Es evidente que se hubiera podido utilizar este artificio en el programa anterior para hacerlo más corto.

Una vez que se haya transferido un bloque completo, no hay más que apuntar hacia la página siguiente en nuestro bloque original y en nuestro bloque de destino. Esto se realiza añadiendo "1" a la parte de orden más alto de la dirección para la fuente y el destino, lo que se efectúa por medio de las dos instrucciones siguientes del programa:

BLOQUE SIG DESDE+1
A+1

Después de haber incrementado el puntero de página, se prueba si hay otro bloque a transferir, o no lo hay, decrementando el contador de bloques contenido en X. Esto se realiza por:

DEX

Si se han transferido todos los bloques, se sale del programa bifurcando a la etiqueta FIN:

BMI FIN

De no ser así, tenemos dos posibilidades: o bien no hemos decrementado hasta "0" o bien hemos llegado exactamente a "0". Si todavía no hemos decrementado a "0", se bifurca a SIGUIENTE:

BNE SIGUIENTE

Si se ha decrementado exactamente a "0", todavía es preciso transferir las palabras especificadas por "RESTO". Esta es la última parte de nuestra transferencia. Esto se realiza mediante:

LDY #RESTO

que carga el índice Y en la cuenta de las palabras que quedan por transferir.

Se bifurca, entonces, a SIGUIENTE:

BNE SIGUIENTE

El lector debe cerciorarse de que, durante este último bucle, o bien se ejecutará la instrucción de bifurcación a SIGUIENTE la primera vez que se volverá a introducir BLOQUE SIG, o bien se saldrá efectiva y realmente del programa. Ello se debe a que el índice X tenía el valor 0 antes de introducir BLOQUE SIG. La tercera instrucción de BLOQUE SIG lo cambiará a −1 y se saldrá a FIN.

Adición de dos bloques

Este ejemplo ilustrará, de manera simple, la utilización de un registro índice para la adición de dos bloques de menos de 256 elementos. A continuación, el programa siguiente utilizará el direccionamiento indirecto indexado para acceder a bloques cuya dirección resida en una posición conocida, pero cuya dirección absoluta real sea desconocida. El programa será como sigue:

```
SUMA BLOQUE   LDY   # NBR - 1      ---CARGAR EL CONTADOR
SIGUIENTE     CLD
              LDA   PTR1,Y         ---LEER ELEMENTOS SIGUIENTES
              ADC   PTR2,Y         ---SUMARLOS
              STA   PTR3,Y            ALMACENAR EL RESULTADO
              DEY                     DECREMENTAR CONTADOR
              BPL   SIGUIENTE         ¿ACABADO?
```

El índice Y se utiliza como un contador de índice y se carga con el número de elementos menos uno. Suponemos que el puntero PTR1 apunta al primer elemento del BLOQUE 1, PTR2 hacia el primer elemento del BLOQUE 2 y PTR3 hacia la zona de destino en donde deben estar almacenados los resultados.

El programa es autoexplicatorio. El último elemento del BLOQUE 1 se lee en el acumulador y luego se añade al último elemento del BLOQUE 2. A continuación, se almacena en la posición adecuada del BLOQUE 3. Se añade el elemento secuencial siguiente y así sucesivamente.

El mismo ejercicio con el empleo de direccionamiento indirecto indexado.

Suponemos, ahora, que las direcciones PTR1, PTR2 y PTR3 no son conocidas a priori. Pero sabemos que están almacenadas en la página 0 en las direcciones LOC1, LOC2 y LOC3. Este es un mecanismo habitual para pasar información entre subprogramas.

El programa correspondiente aparece así:

```
SUMA BLOQUE   LDY   #NBR-1
SIGUIENTE     CLC
              LDA   (LOC1), Y
              ADC   (LOC2), Y
              STA   (LOC3), Y
              DEY
              BPL   SIGUIENTE
```

La correspondencia entre este nuevo programa y el anterior debe ser evidente ahora. Ilustra claramente el interés del empleo del direccionamiento indirecto indexado cuando la dirección absoluta no es conocida en el momento en que se escribe el programa, pero se conoce la posición de esta dirección. Se puede observar que los dos programas tienen exactamente el mismo número de instrucciones. Un ejercicio interesante sería ahora determinar cuál de los dos es el más rápido en la ejecución.

Ejercicio 5.2: *Calcular el número de bytes y el número de ciclos para cada uno de estos dos programas, con el empleo de las tablas del apéndice D.*

RESUMEN

Se ha presentado una descripción completa de los modos de direccionamiento. Se ha demostrado que el 6502 ofrece la mayor parte de los posibles mecanismos de direccionamiento y se han analizado sus características. Finalmente, se han presentado varios programas de aplicación para poner de manifiesto el valor de cada uno de los mecanismos de direccionamiento. La programación del 6502 requiere una comprensión de estos mecanismos.

EJERCICIOS

5.3: *Escribir un programa que sume los 10 primeros bytes de una tabla situada en la dirección "BASE". El resultado tendrá 16 bits (es el cálculo de una suma de control).*

5.4: *¿Puede tratar el mismo problema sin utilizar el direccionamiento indexado?*

5.5: *Invierta el orden de los 10 bytes de esta tabla. Almacene el resultado en la dirección "INVER"*

5.6: *Busque, en la misma tabla, el elemento más grande. Almacénelo en la posición de memoria "MAX".*

5.7: *Sume los elementos correspondientes de tres tablas cuyas direcciones de base sean BASE 1, BASE 2 y BASE 3. La longitud de estas tablas se almacenan en página cero en la dirección "LONGITUD".*

6 Técnicas de entradas/salidas

INTRODUCCIÓN

Hasta ahora hemos aprendido cómo intercambiar información entre la memoria y los diversos registros del procesador. Hemos aprendido a trabajar con los registros y a utilizar diversas instrucciones para manipular los datos. Ahora hemos de aprender a establecer comunicación con el mundo exterior. Esto es lo que se denomina "entradas/salidas".

Las entradas se refieren a la captación de datos a partir de periféricos exteriores (teclado, disco, o sensor físico).

Las salidas se refieren a la transferencia de datos desde el microprocesador, o la memoria, a dispositivos externos, tales como una impresora, un TRC, un disco o relés y sensores accionadores.

Vamos a proceder en dos etapas. En primer lugar aprenderemos a efectuar las operaciones de entradas/salidas requeridas por los periféricos habituales. A continuación aprenderemos a trabajar con varias unidades de entrada/salida de forma simultánea; es decir, a *sincronizarlas*. Esta segunda parte cubrirá, en particular, la elección entre *escrutinio* o *interrupciones*.

ENTRADAS/SALIDAS

En esta sección aprenderemos a detectar o a generar señales simples, tales como impulsos. A continuación estudiaremos técnicas para forzar o medir temporizaciones correctas. Estaremos, pues, preparados para realizar entradas/salidas más complejas, como las transferencias rápidas en serie o en paralelo.

6 Técnicas de entradas/salidas

INTRODUCCIÓN

Hasta ahora hemos aprendido cómo intercambiar información entre la memoria y los diversos registros del procesador. Hemos aprendido a trabajar con los registros y a utilizar diversas instrucciones para manipular los datos. Ahora hemos de aprender a establecer comunicación con el mundo exterior. Esto es lo que se denomina "entradas/salidas".

Las entradas se refieren a la captación de datos a partir de periféricos exteriores (teclado, disco, o sensor físico).

Las salidas se refieren a la transferencia de datos desde el microprocesador, o la memoria, a dispositivos externos, tales como una impresora, un TRC, un disco o relés y sensores accionadores.

Vamos a proceder en dos etapas. En primer lugar aprenderemos a efectuar las operaciones de entradas/salidas requeridas por los periféricos habituales. A continuación aprenderemos a trabajar con varias unidades de entrada/salida de forma simultánea; es decir, a *sincronizarlas.* Esta segunda parte cubrirá, en particular, la elección entre *escrutinio* o *interrupciones.*

ENTRADAS/SALIDAS

En esta sección aprenderemos a detectar o a generar señales simples, tales como impulsos. A continuación estudiaremos técnicas para forzar o medir temporizaciones correctas. Estaremos, pues, preparados para realizar entradas/salidas más complejas, como las transferencias rápidas en serie o en paralelo.

Generación de una señal

En el caso más simple, un dispositivo de salida será conectado, o desconectado, por medio del ordenador. Para cambiar el estado del periférico, el programador sólo tendrá que cambiar un nivel lógico de "1" a "0" o de "0" a "1".

Supongamos que el bit 0 de un registro denominado "SALIDA 1" esté conectado a un relé exterior. Para la activación, simplemente escribiremos un "1" en la posición de bit adecuada del registro. Supongamos que SALIDA 1 representa la dirección de este registro de salida en nuestro sistema. El programa que activará el relé es:

```
ACTIVACIÓN   LDA   #%00000001
             STA   SALIDA 1
```

Hemos supuesto que el estado de los otros 7 bits del registro SALIDA 1 carece de importancia y significado. Sin embargo, ello no suele ser lo más normal. Estos bits podrían conectarse a otros relés. Mejoremos, pues, este programa simple. Deseamos activar el relé sin cambiar el estado de ningún bit del registro. Suponemos que el registro puede ser objeto de lectura o de escritura.

Nuestro programa se hace, entonces:

```
ACTIVACIÓN  LDA  SALIDA 1     LEER EL CONTENIDO DE
                              SALIDA 1
            ORA  #%00000001   FORZAR A "1" EL BIT 0
            STA  SALIDA 1
```

El programa lee primero el contenido de la posición SALIDA 1 y luego realiza una función OR inclusiva con su contenido. Ello cambia solamente a "1" el bit en la posición 0 y deja intacto el resto del registro (para más detalles sobre la instrucción ORA, consúltese el capítulo 4). Esto se ilustra por medio de la figura 6-1.

Impulsos

Se genera un *impulso* exactamente como en el caso de un *nivel*, según se vio anteriormente. En primer lugar, se pone a "1" un bit de salida y, luego, se pone a "0". Ello da lugar a la generación de un impulso, tal como se ilustra en la figura 6-2. No obstante, en esta ocasión se plantea un problema adicional: se debe generar un impulso de duración correcta. Estudiemos, pues, cómo generar un retardo calculado.

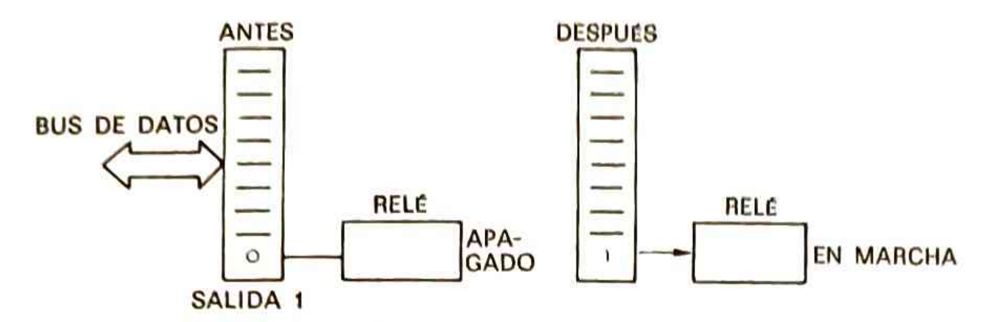


Figura 6-1 Activación de un relé.

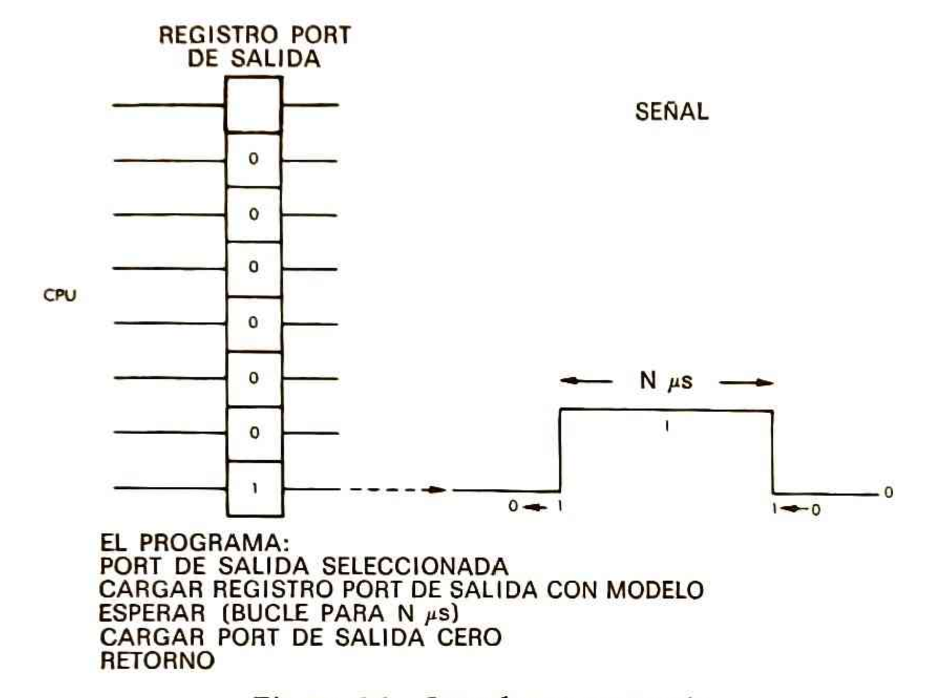


Figura 6-2 Impulso programado.

Generación y medida de un retardo

Un retardo puede generarse por medio de métodos de software o de hardware. Estudiaremos ahora cómo hacerlo mediante un programa y, más adelante, cómo puede realizarse con un contador de hardware llamado generador de intervalos de tiempo programables (PIT).

Los retardos programados se obtienen por contaje. Se carga un registro de contador con un valor y luego se decrementa. El programa efectúa un bucle en sí mismo y continúa decrementando hasta que el contador alcance el valor "0". El tiempo total requerido por este proceso proporcionará el retardo requerido. Por ejemplo, generemos un retardo de 36 microsegundos:

```
RETARDO     LDY   #07          EL CONTADOR ES Y
SIGUIENTE   DEY                DECREMENTAR
            BNE   SIGUIENTE    TEST
```

Este programa carga el registro de índice Y con el valor 7. La siguiente instrucción decrementa Y y la siguiente hará que se produzca una bifurcación a SIGUIENTE, en tanto que Y no se haya decrementado hasta "0".

Cuando Y se decrementa finalmente a "0", el programa saldrá del bucle y ejecutará las instrucciones que puedan seguir. La lógica de este programa es simple y se ilustra en el diagrama de flujo de la figura 6-3.

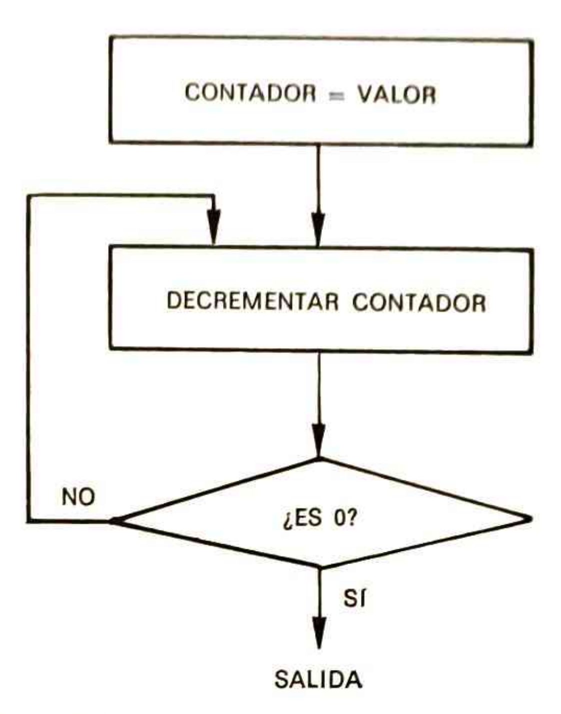


Figura 6-3 Diagrama de flujo de un retardo.

Calculemos el retardo efectivo que se obtendrá mediante este programa. Consultando el apéndice D del final del libro se encontrará el número de ciclos correspondiente a cada una de las instrucciones:

LDY, en el modo inmediato, requiere 2 ciclos. DEY utilizará 2 ciclos. Finalmente, BNE requerirá 3 ciclos. Al examinar en la tabla el número de ciclos requeridos por BNE se constata que hay 3 posibilidades. Si la bifurcación no tiene lugar, BNE no requiere más que 2 ciclos. Si se produce la bifurcación, lo que es el caso normal durante el bucle, es preciso un ciclo más. Finalmente, si se cruza una frontera de página, se necesita un ciclo suplementario. En este caso, se supondrá que no se franquea frontera de página.

La duración es, pues, de dos ciclos para la primera instrucción, más cinco ciclos para las dos siguientes multiplicado por el número de veces en que se ejecutará el bucle, menos un ciclo ya que en la última BNE no existe bifurcación:

$$\text{Retardo} = 2 + (5 \times 7) - 1 = 36$$

Suponiendo un tiempo de ciclo de 1 microsegundo, este retardo programado será de 36 microsegundos.

En este ejemplo simple podemos constatar, que la resolución más fina con la que se puede ajustar este retardo es de 2 microsegundos. El retardo mínimo es de 2 microsegundos.

Ejercicio 6.1: *¿Cuál es el retardo máximo que se puede obtener con estas tres instrucciones? ¿Puede modificar el programa para obtener un retardo de un microsegundo?*

Ejercicio 6.2: *Modificar el programa para obtener un retardo de unos 100 microsegundos.*

Si se desea obtener un retardo más largo, una solución sencilla es añadir instrucciones en el programa entre DEY y BNE. La manera más simple de hacerlo es incluir instrucciones NOP (la instrucción NOP no hace nada, pero dura 2 ciclos).

Retardos más largos

La generación por software de retardos más largos puede conseguirse con el empleo de un contador más grande. Se pueden utilizar dos registros internos, o mejor dos palabras en la memoria, para mantener un contaje de 16 bits. Para simplificar, supongamos que el contaje inferior sea "0". El byte inferior se cargará con "255" (el contaje máximo) y luego se introducirá en un bucle de decrementación. Siempre que se decremente a "0", el byte superior del contador se decrementará en 1. Cuando el byte superior se decrementa al valor "0", se terminará el programa. Si se requiere más precisión en la generación de retardos, el contaje más bajo puede tener un valor no nulo. En este caso, se escribirá el programa tal como se explicó y se añadirá al final el programa de generación de retardos de tres líneas anteriormente descrito.

Naturalmente, se podrían generar retardos todavía más largos con el empleo de más de dos palabras. Esto es análogo a la forma en que funciona un indicador kilométrico en un automóvil. Cuando la rueda más a la derecha pasa de "9" a "0", la siguiente rueda a la izquierda se incrementa en 1. Este es el principio general del contaje con unidades discretas.

Sin embargo, la principal objeción es que, cuando se cuentan retardos largos, el microprocesador no hace nada más durante centenas de milisegundos o incluso segundos. Si el ordenador no tiene nada que hacer, ello es perfectamente aceptable. Sin embargo, en el caso general, es preciso que el microordenador quede disponible para otras tareas, de modo que no se rea-

licen normalmente los retardos largos por software. De hecho, incluso los retardos cortos programados pueden ser inconvenientes en un sistema, si es preciso proporcionar tiempos de respuesta garantizados en ciertas situaciones. Entonces, se debe obtener el retardo por hardware. Además, si se utilizan las interrupciones, la exactitud de los retardos corre el riesgo de perderse si puede interrumpirse el bucle de contaje.

Ejercicio 6.3: *Escribir un programa para obtener un retardo de 100 ms (para un teletipo).*

Retardos por hardware

Los retardos por hardware se obtienen con la ayuda de un generador de intervalos de tiempo programable o, dicho brevemente, mediante un "temporizador". Un registro del temporizador está cargado con un valor. La diferencia es que, esta vez, el temporizador decrementará automáticamente este contador periódicamente. El período suele ser ajustable o seleccionable por el programador. Cuando el temporizador se decremente hasta "0", enviará normalmente una interrupción al microprocesador. También puede posicionar un bit de estado que puede probarse periódicamente por el ordenador. La utilización de las interrupciones se explicará más adelante en este mismo capítulo.

Otros modos de funcionamiento del temporizador pueden permitir la partida desde "0" y medir la duración de la señal o incluso contar el número de impulsos recibidos. Cuando funciona como un generador de intervalos, se dice que el temporizador funciona en modo *monoestable*. Cuando cuenta impulsos, se dice que funciona en un modo de *contador*. Algunos dispositivos temporizadores pueden comprender también registros múltiples y un cierto número de opciones que pueden seleccionarse mediante programa. Es el caso, por ejemplo, de los temporizadores contenidos en el 6522, dispositivo integrado de E/S descrito en el siguiente capítulo.

Detección de impulsos

El problema de la detección de impulsos es el inverso de la generación de impulsos, con una dificultad adicional: mientras que los impulsos de salida son generados bajo control del programa, los impulsos de entrada llegan *asíncronamente* con respecto al programa. Para detectar un impulso, se puede utilizar dos métodos: *escrutinio* e *interrupciones*. Las interrupciones se tratarán más adelante en este capítulo.

Consideremos la técnica de escrutinio. Con el empleo de esta técnica, el programa lee continuamente el valor de un registro de entrada dado, compro-

bando una posición de bit, por ejemplo, el bit 0. Se supondrá que el bit 0 es originalmente "0". Siempre que se reciba un impulso, este bit tomará el valor "1". El programa controla constantemente el bit 0 hasta que tome el valor "1". Cuando se encuentre un "1", se ha detectado el impulso.

El programa correspondiente es el siguiente:

```
ESCRUTINIO          LDA    #$01
DE NUEVO            BIT    ENTRADA
                    BEQ    DE NUEVO
ASI SUCESIVAMENTE   ...
```

Por el contrario, supongamos que la línea de entrada esté normalmente en "1" y que se desea detectar un "0". Este es el caso normal para detectar un bit START (de partida) cuando se controla continuamente una línea conectada a un teletipo. El programa es el siguiente:

```
ESCRUTINIO   LDA    #$01
SIGUIENTE    BIT    ENTRADA
             BNE    SIGUIENTE
START        ...
```

Medida de la duración

La medida de la duración del impulso puede hacerse de la misma forma que para calcular la duración de un impulso en salida. Se puede utilizar una técnica de hardware o bien una técnica de software. Cuando se mide un impulso por software se incrementa un contador regularmente y luego se verifica la presencia del impulso. Si el impulso está todavía presente, el programa realiza un bucle sobre sí mismo. Cuando desaparece el impulso, se utiliza el contaje contenido en el registro de contador para calcular la duración efectiva del impulso. El programa correspondiente es el siguiente:

```
DURACIÓN     LDX   #0          PONER A CERO EL CONTADOR
             LDA   #$01        SE CONTROLA EL BIT 0
DE NUEVO     BIT   ENTRADA
             BEQ   DE NUEVO
MÁS LARGO    INX
             BIT   ENTRADA
             BNE   MÁS LARGO
```

Naturalmente, suponemos que la duración máxima del impulso no llevará consigo el desbordamiento de capacidad del registro X. Si este fuera el

caso, sería preciso hacer más largo el programa para tenerlo en cuenta (o si no, sería un error de programación).

Puesto que ahora sabemos cómo detectar y generar impulsos, vamos a adquirir más grandes cantidades de datos. Se distinguirán dos casos: datos en serie y datos en paralelo. A continuación aplicaremos estos conocimientos a los dispositivos de entrada/salida existentes.

TRANSFERENCIA DE PALABRAS EN PARALELO

En este caso se supone que los 8 bits de datos a transferir están disponibles en paralelo en la dirección "ENTRADA". El microprocesador debe leer la palabra de datos en esta posición, siempre que una palabra de estado indique que es válida. Se supondrá que la información de estado está contenida en el bit 7 de la dirección "ESTADO". Vamos a escribir un programa que

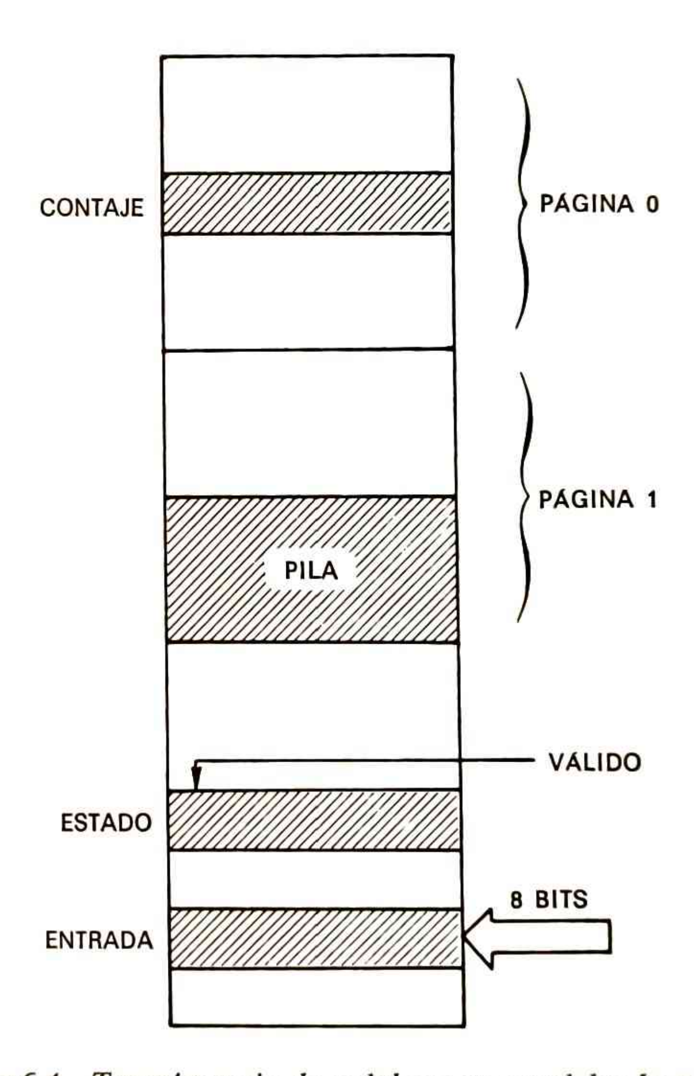


*Figura 6-4 Transferencia de palabras en paralelo: la **memoria**.*

leerá y conservará automáticamente cada palabra de datos tal como llega. Para simplificar, supondremos que el número de palabras a leer se conoce de antemano y está contenido en la dirección "CONTAJE". Si esta información no estuviera disponible, se probaría con respecto a un carácter, denominado *carácter de parada,* tal como un *borrado* o quizás el carácter "*" Ya hemos aprendido a hacerlo.

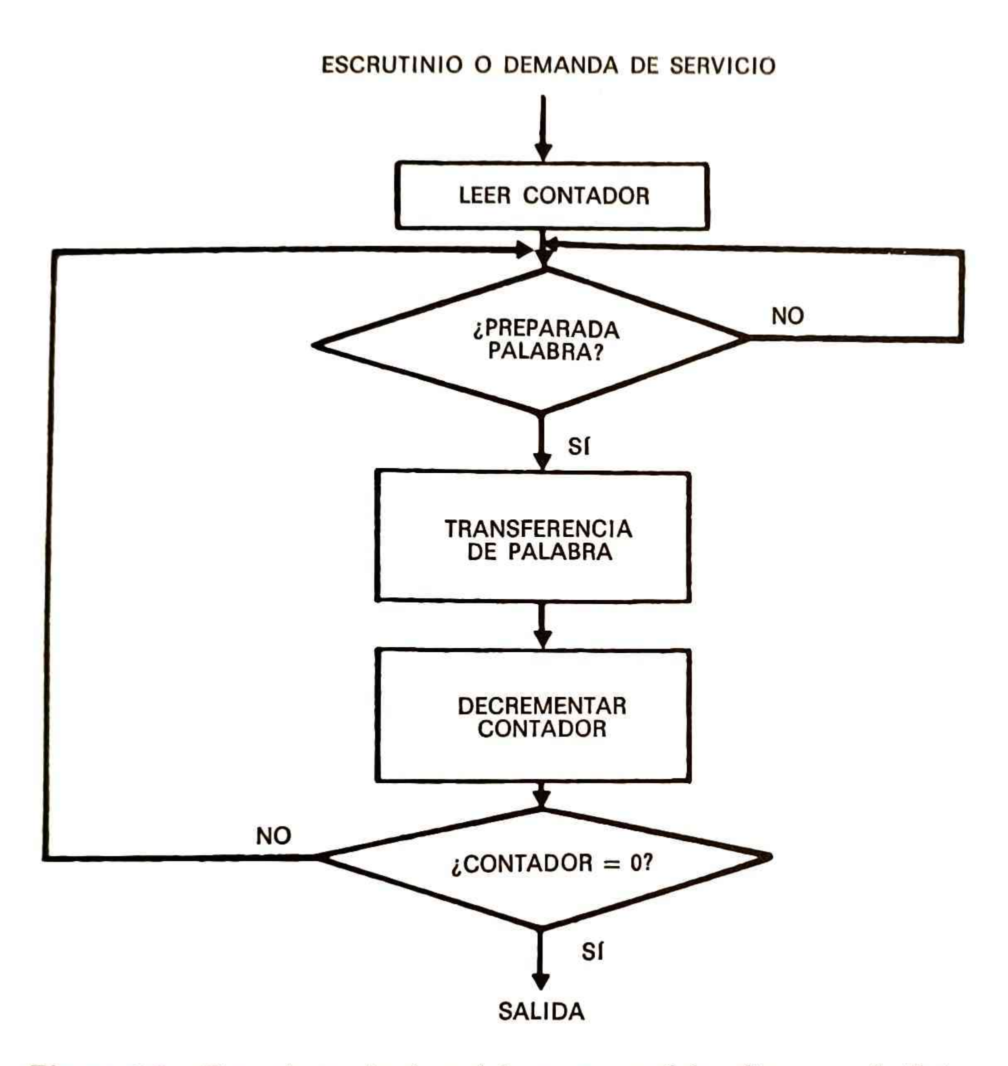


Figura 6-5 Transferencia de palabras en paralelo: diagrama de flujo.

En la figura 6-5 se ilustra el diagrama de flujo correspondiente. Es muy sencillo. Probamos la información de estado hasta que se haga "1" indicando que está preparada una palabra. Entonces leemos esta última y la conservamos en una posición de memoria adecuada. Decrementamos el contador y luego comprobamos si se ha decrementado a "0". De ser así, hemos acabado; de no serlo, leemos la palabra siguiente.

El programa que realiza este algoritmo es el siguiente:

```
PARAL     LDX   CONTAJE   CONTADOR
CONTROL   LDA   ESTADO    EL BIT 7 ES "1" SI EL DATO
                          ES VÁLIDO
          BPL   CONTROL   ¿DATO VÁLIDO?
          LDA   ENTRADA   LEER EL DATO
          PHA             CONSERVARLO EN LA PILA
          DEX
          BNE   CONTROL
```

Las dos primeras instrucciones del programa leen la información de estado y hacen que se efectúe un bucle en tanto que el bit 7 del registro de estado es "0" (es el bit de signo, es decir, el bit N).

```
CONTROL   LDA   ESTADO
          BPL   CONTROL
```

Cuando BPL es negativo, el dato es válido y se puede leer:

```
LDA   ENTRADA
```

La palabra se ha leído ahora a partir de la dirección ENTRADA en donde estaba y debe conservarse. Suponiendo que el número de palabras a transferir es bastante pequeño, se utiliza un:

```
PHA
```

Si la pila corre el riesgo de estar llena o si el número de palabras a transferir es grande, no se podrían poner en la pila y sería preciso almacenarlas en una zona de memoria dada, con el empleo, por ejemplo, de una instrucción indexada. Sin embargo, ello requeriría una instrucción adicional para incrementar o decrementar el registro de índice. PHA es más rápida.

La palabra de datos se ha leído ahora y se ha conservado. Decrementaremos simplemente el contador de palabras y comprobaremos si hemos acabado:

```
DEX
BNE   CONTROL
```

Continuamos realizando el bucle hasta que el contador llegue a "0". Este programa de 6 instrucciones puede considerarse como una *referencia de prestaciones* ("benchmark"), es decir, como un programa para la medida de prestaciones. Se trata de un programa cuidadosamente optimizado que

se ha concebido para comprobar las capacidades de un procesador dado en una situación específica. Las transferencias en paralelo constituyen una situación característica de estas circunstancias operativas. Este programa se ha concebido con miras a una eficacia y a una velocidad máximas. Calculemos ahora la velocidad de transferencia máxima permitida por este programa. Supondremos que el contaje está en la página 0. La duración de cada instrucción se obtiene examinando la tabla del apéndice D. Se encuentra:

			CICLOS	
	LDX	CONTAJE	3	
CONTROL	LDA	ESTADO	4	
	BPL	CONTROL	2/3	(INSATISFACTORIO/ SATISFACTORIO)
	LDA	ENTRADA	4	
	PHA		3	
	DEX		2	
	BNE	CONTROL	2/3	(INSATISFACTORIO/ SATISFACTORIO)

El tiempo de ejecución mínimo se obtiene suponiendo que el dato está siempre disponible cuando se prueba el ESTADO. Dicho de otro modo, el primer BPL se supondrá que es insatisfactorio secuencialmente cada vez. El tiempo es, entonces:

$$3 + (4+2+4+3+2+3) \times \text{CONTAJE}$$

Si se desprecian los tres primeros microsegundos necesarios para inicializar el registro de contador, el tiempo utilizado para transferir una palabra es 18 microsegundos.

La tasa máxima de transferencia de datos es, pues:

$$\frac{1}{18(10^{-6})} = 55 \text{ K bytes por segundo.}$$

Ejercicio 6.4: *Supóngase que el número de palabras a transferir es mayor que 256. Modificar el programa consecuentemente y determinar su influencia sobre la tasa máxima de transferencia de datos.*

Hemos aprendido a efectuar transferencias en paralelo rápidas. Consideremos un caso más complejo.

TRANSFERENCIA EN SERIE

Una entrada en serie es aquella en la que los bits de información ("0" o "1") llegan sucesivamente en una línea. Dichos bits pueden llegar a intervalos regulares. Este proceso suele denominarse transmisión *síncrona*. O bien, pueden llegar en grupos ("ráfagas") de datos a intervalos aleatorios, lo que se llama transmisión *asíncrona*. Desarrollaremos un programa que puede trabajar en ambos casos. El principio de la adquisición de datos secuenciales es sencillo: vigilaremos una línea de entrada, que se supondrá que es la línea "0". Cuando un bit de datos se detecta en esta línea, lo leeremos y lo haremos entrar por desplazamiento en un registro de memorización. Cuando se hayan ensamblado 8 bits, conservaremos el byte de datos en la memoria y ensamblaremos el siguiente.

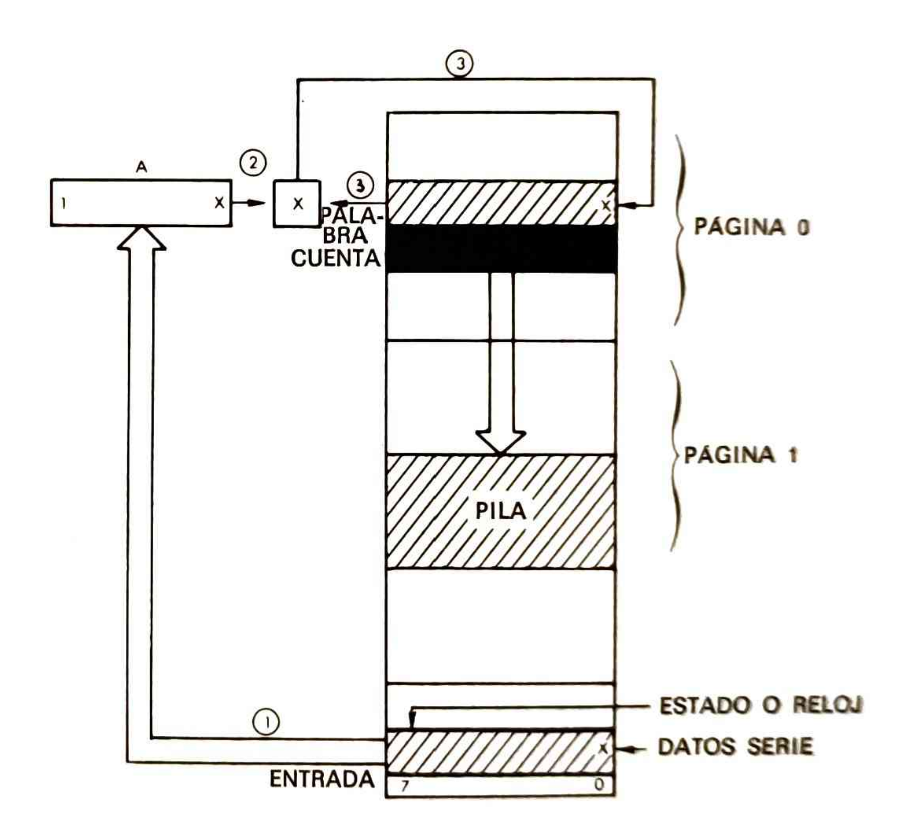


Figura 6-6 Conversión serie a paralelo.

Para simplificar, supondremos conocido, de antemano, el número de bytes a recibir. De no ser así, podríamos, por ejemplo, probar la llegada de un carácter especial de interrupción y detener la transferencia en este momento. Hemos aprendido a hacerlo así. En la figura 6-7 se ilustra el diagrama de flujo de este programa. El programa correspondiente es:

```
SERIE   LDA #$00
        STA PALABRA
BUCLE   LDA ENTRADA EL BIT 7 ES EL ESTADO, EL BIT "0"
                    ES EL DATO
        BPL BUCLE   ¿RECIBIDO BIT?
        LSR A       DESPLAZARLO A C
        ROL PALABRA CONSERVAR BIT EN MEMORIA
        BCC BUCLE   CONTINUAR SI ACARREO = "0"
        LDA PALABRA
        PHA         CONSERVAR BYTE ENSAMBLADO
        LDA #$01    INICIALIZAR CONTADOR DE BITS
        STA PALABRA
        DEC CONTAJE DECREMENTAR CONTADOR PALABRA
        BNE BUCLE   ENSAMBLAR LA PALABRA SIGUIENTE
```

Este programa se ha concebido para ser eficaz gracias a técnicas nuevas que vamos a explicar a continuación (fig. 6-6).

Los convenios son los siguientes: la posición de memoria CONTAJE se supone que contiene el número de palabras a transferir. La posición de memoria PALABRA servirá para ensamblar 8 bits entrantes consecutivos. La dirección ENTRADA designa un registro de entrada. Se supone que la posición 7 de este registro es un indicador de estado o un bit de reloj. Cuando vale "0", el dato no es válido. Cuando es "1", el dato es válido. El dato propiamente dicho aparece en el bit 0 de esta misma dirección. En muchos casos, la información de estado aparece en un registro diferente del de datos. Entonces será una tarea sencilla modificar el programa consecuentemente. Además, supondremos que el primer bit de datos a recibir por este programa está garantizado que es un "1". Indica que siguen los datos propiamente dichos. Si este no fuera el caso, veremos más adelante una modificación evidente para tenerlo en cuenta. El programa corresponde exactamente al diagrama de flujo de la figura 6-7. Las primeras líneas del programa forman un bucle de espera que prueba si está dispuesto un bit. Para determinarlo, leemos el registro de entrada y luego probamos el bit de signo (N). En tanto que este bit sea "0", la instrucción BPL tendrá resultados satisfactorios y se vuelve al bucle. Cuando el bit de estado (o de reloj) se haga "1", la BPL ya no producirá bifurcación y se ejecutará la instrucción que sigue.

Recuérdese que BPL significa "bifurcación si es más", es decir, cuando el bit 7 (bit de signo) es "0". Esta secuencia de instrucciones corresponde a la flecha 1 de la figura 6-6.

En este momento, el acumulador contiene un "1" en la posición 7 y el bit de dato propiamente dicho está en la posición 0. El primer bit de dato será un "1". Pero los siguientes pueden ser "0" o "1". Deseamos, ahora,

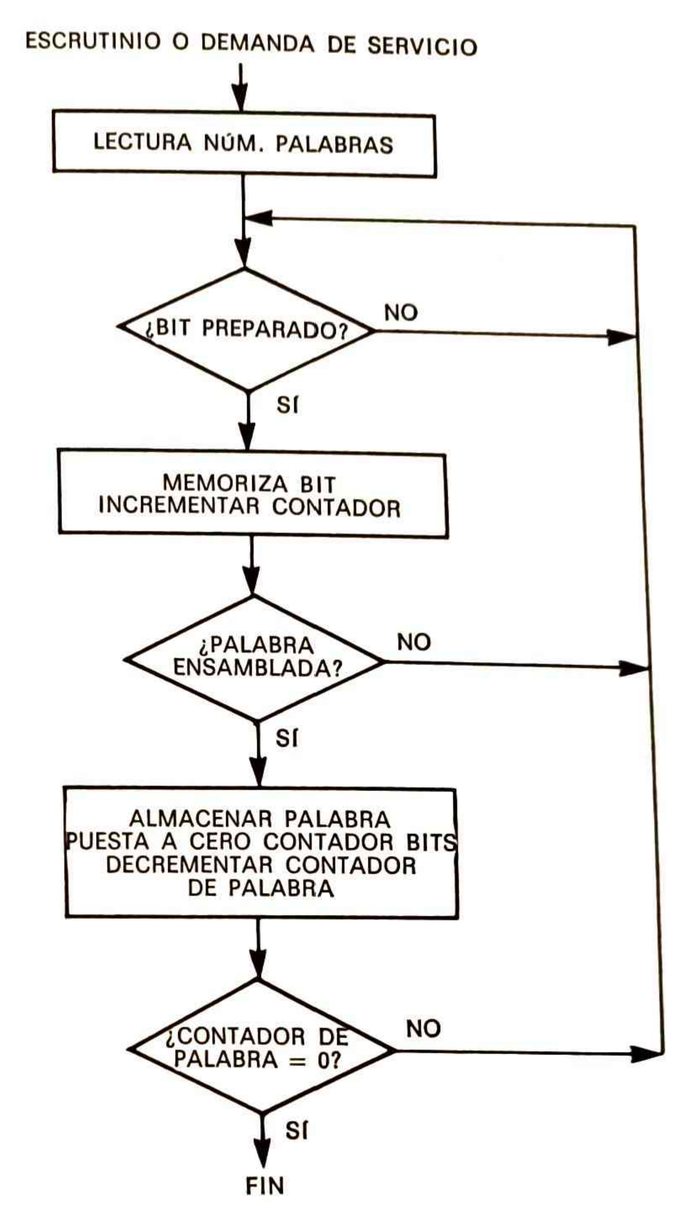


Figura 6-7 Transferencia de bits en serie: diagrama de flujo.

salvaguardar el bit de dato que se ha recogido en la posición 0. La instrucción:

LSR A

desplaza en una posición a la derecha el contenido del acumulador. Esto lleva el bit más a la derecha de A, es decir nuestro bit de dato, al acarreo. Vamos a conservarlo en la posición de memoria PALABRA (esto se ilustra por las flechas 2 y 3 de la figura 6-6):

ROL PALABRA

El efecto de esta instrucción es transferir el bit de acarreo a la posición más a la derecha de la dirección PALABRA. Al mismo tiempo, el bit más a la izquierda de PALABRA cae en el bit de acarreo (si se tienen dudas sobre el funcionamiento de la rotación, consúltese el capítulo 4). Es importante recordar que una rotación tiene por efecto salvaguardar el bit de acarreo, en este caso en la posición más a la derecha, y también volver a copiar el valor del bit 7 en el bit de acarreo.

Aquí es un "0" que caerá en el acarreo. La instrucción siguiente:

```
BCC    BUCLE
```

prueba el acarreo y retorna a BUCLE en tanto que el acarreo sea nulo. Este es nuestro contador de bits automático. Se puede ver fácilmente que, como consecuencia de la primera ROL, PALABRA contendrá "00000001". Ocho desplazamientos más tarde, llegará finalmente el "1" al bit de acarreo y hará que se interrumpan las bifurcaciones. Es una manera ingeniosa de realizar un contador de bucle automático sin tener que dedicar una instrucción a decrementar un registro de índice. Esta técnica se utiliza para acortar el programa y aumentar sus prestaciones.

Cuando BCC ya no lleva a una bifurcación, es que 8 bits han acabado por ensamblarse en la posición de PALABRA. Este valor debe salvaguardarse en memoria, lo que se realiza por las instrucciones siguientes (flecha 4 en la figura 6-6):

```
LDA    PALABRA
PHA
```

En este caso conservamos la PALABRA de datos (8 bits) en la pila. La salvaguardia en la pila sólo es posible si hay bastante espacio en la misma. Con tal que se cumpla esta condición, es la manera más rápida de conservar una palabra en la memoria. El puntero de pila se actualiza automáticamente. Si no utilizáramos la pila, deberíamos emplear una instrucción para actualizar un puntero hacia la memoria. Podríamos, de forma equivalente, utilizar el direccionamiento indexado, pero ello traería consigo el incremento, o decremento, de índice, lo que consumiría tiempo.

Cuando se haya conservado la primera PALABRA de datos, ya no hay ninguna garantía de que el primer bit de datos a llegar sea un "1", pudiendo ser cualquiera. Por consiguiente, debemos reinicializar el contenido de PALABRA en "00000001" para poder continuar su empleo como contador de bits. Ello se efectúa por medio de las dos instrucciones siguientes:

```
LDA    #$01
STA    PALABRA
```

Finalmente, decrementaremos el contaje de palabras, puesto que se ha ensamblado una palabra y probaremos si hemos alcanzado el final de la transferencia. Ello se realiza por medio de las dos instrucciones siguientes:

```
DEC   CONTAJE
BNE   BUCLE
```

El programa anterior se ha concebido para ser rápido, con el fin de poder captar una serie de bits de datos rápida. Una vez terminado el programa, es recomendable, por supuesto, leer en la pila las palabras que se han conservado en ella y transferirlas a cualquier lugar en la memoria. Ya hemos aprendido a efectuar una tal transferencia de bloque en el capítulo 2.

Ejercicio 6.5: *Calcular la velocidad máxima a la que este programa será capaz de leer bits en serie. Para calcular esta velocidad, ha de suponerse que las direcciones PALABRA y CONTAJE se mantienen en página 0. Asimismo, se supondrá que todo el programa reside en una misma página. Examine el número de ciclos requeridos por cada instrucción en la tabla del apéndice D y calcule el tiempo transcurrido durante la ejecución del programa. Para calcular el tiempo utilizado por un bucle, basta multiplicar la duración total de este bucle (expresada en microsegundos) por el número de veces que se ejecutará. Además, para calcular la velocidad máxima suponga que un bit de dato estará dispuesto cada vez que se pruebe el registro de entrada.*

Este programa es más difícil de comprender que los anteriores. Examinémoslo de nuevo (fig. 6-6) con más detalle, y tratemos de las soluciones de compromiso. Llega un bit de dato de vez en cuando a la posición 0 de "ENTRADA". Puede, por ejemplo, tener tres "1" sucesivos. Debemos, pues, distinguir los bits sucesivos que llegan. Esta es la función de la señal de "reloj".

La señal de reloj (o de ESTADO) nos indica que el bit a la entrada es válido.

Antes de leer un bit, probemos primero el bit de estado. Si el estado es "0", tendremos que esperar. Si es "1", entonces el bit dato es válido. Suponemos, en este caso, que la señal de estado está conectada al bit 7 del registro de ENTRADA.

Ejercicio 6.6: *¿Puede explicar por qué se utiliza el bit 7 para el reloj y el bit 0 para el dato?*

Una vez que hayamos recogido un bit de dato, deseamos conservarlo en

un lugar seguro y luego desplazarlo a la izquierda con el fin de poder captar el bit siguiente.

Lamentablemente, en este programa se utiliza el acumulador para leer y probar, a la vez, el reloj y el dato. Si acumuláramos los datos en el acumulador, el bit 7 sería borrado por el bit de estado.

Ejercicio 6.7: *¿Puede sugerir una manera de probar el estado sin borrar el contenido del acumulador (con la ayuda de una instrucción especial)? Si ello puede hacerse, ¿podríamos utilizar el acumulador para realizar la acumulación de los bits sucesivos que llegan?*

Ejercicio 6.8: *Volver a escribir el programa utilizando el acumulador para almacenar los bits entrantes. Comparar con el anterior en lo que respecta a velocidad y al número de instrucciones.*

Consideremos otras dos variaciones posibles:

Hemos supuesto que, en nuestro caso particular, el primer bit a entrar sería una señal especial, garantizada como un "1". Pero, en el caso general, puede tener cualquier valor.

Ejercicio 6.9: *Modifique el programa anterior, suponiendo que cualquier primer bit forma parte de los datos válidos (y no tiene, pues, que desecharse) y puede tener el valor "0" o "1" (Recomendación: Nuestro "contador de bits" debe seguir funcionando correctamente, si se inicializa con el valor correcto).*

Finalmente, hemos estado conservando la PALABRA ensamblada en la pila para ganar tiempo. Naturalmente, podríamos conservarla en una zona de memoria especificada.

Ejercicio 6.10: *Modifique el programa anterior para almacenar la PALABRA ensamblada en la zona de memoria que comienza en BASE.*

Ejercicio 6.11: *Modifique el programa anterior de forma que la transferencia se interrumpa cuando se detecte, en el flujo de entrada, el carácter "S".*

Alternativa de hardware

Como es habitual para la mayor parte de los algoritmos de entrada/salida, es posible realizar el procedimiento anterior por medio de hardware. El dispositivo que lo hace se denomina UART. Acumula automáticamente

los bits. Pero cuando se desea reducir el número de componentes, se utilizará preferiblemente un programa o una de sus variantes.

Ejercicio 6.12: *Modifique el programa suponiendo que el dato se presenta en el bit 0 de la posición ENTRADA, mientras que la información del estado está disponible en el bit 0 de la dirección ENTRADA + 1.*

RESUMEN DE ENTRADAS/SALIDAS BÁSICAS

Ya hemos aprendido a efectuar las operaciones de entradas/salidas elementales, así como a tratar un flujo de datos en paralelo o bits en serie. Ahora estamos en condiciones de comunicar con periféricos reales.

COMUNICACIÓN CON PERIFÉRICOS

Para intercambiar datos con los periféricos tendremos que cerciorarnos de que los datos están disponibles cuando deseamos proceder a su lectura o si el periférico está en condiciones de aceptar datos cuando deseamos enviarlos. Pueden utilizarse dos procedimientos: diálogo e interrupciones. Estudiemos primero el diálogo.

Diálogo

El diálogo ("handshaking") suele utilizarse para comunicarse entre dos dispositivos asíncronos, es decir, entre dos dispositivos que no estén sincronizados automáticamente. Por ejemplo, si queremos enviar una palabra a una impresora en paralelo, ante todo debemos cerciorarnos de que el buffer de entrada de la impresora está disponible. Vamos, pues, a preguntar a la

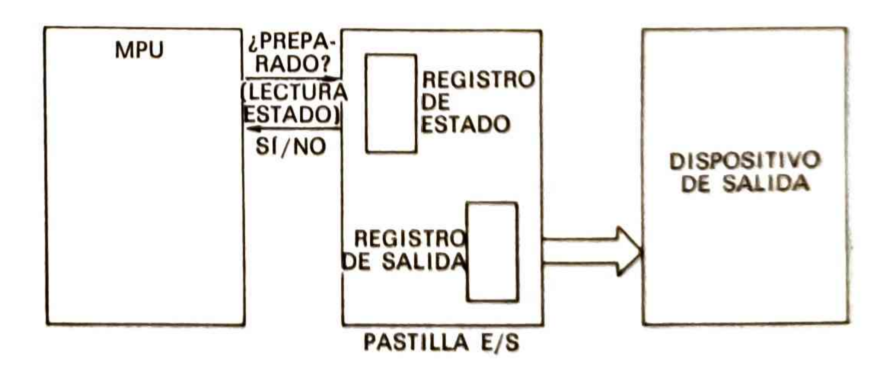


Figura 6-8 Establecimiento de una comunicación o diálogo (salida).

impresora "¿estás preparada"? La impresora responderá "sí" o "no". Si no está preparada, esperaremos. Si está preparada, enviaremos los datos (figura 6-8).

Por el contrario, antes de leer datos desde una unidad de entrada, verificaremos si el dato es válido. Preguntaremos ¿"es válido el dato"? Y el periférico nos responderá "sí" o "no". La respuesta puede proporcionarse por bits de estado o por otros medios (fig. 6-9).

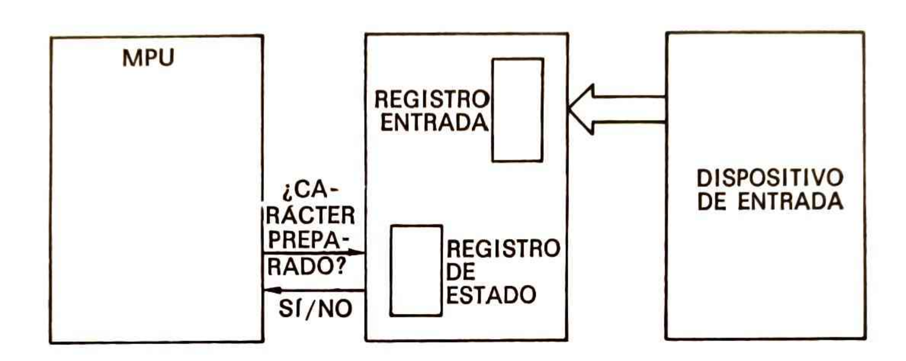


Figura 6-9 Establecimiento de una comunicación o diálogo (entrada).

En pocas palabras, cada vez que queramos intercambiar informaciones con alguien que sea independiente y pudiera estar realizando otra cosa en ese momento, es preciso cerciorarnos de que está dispuesto a comunicarse, de que está en condiciones de establecer un diálogo. El intercambio de datos vendrá luego. Este es el procedimiento que suele utilizarse en la comunicación con los periféricos.

Expliquemos ahora este procedimiento con un ejemplo sencillo.

Envío de un carácter a una impresora

Se supondrá que el carácter está contenido en la posición de memoria CAR. El programa para imprimirlo es el siguiente:

```
IMPCAR   LDX   CAR      LEER EL CARÁCTER
ESPERA   LDA   ESTADO   EL BIT 7 CONTIENE EL INDICADOR
                        "PREPARADA".
         BPL   ESPERA
         TXA
         STA   IMPRD
```

El registro X se carga primero a partir de la memoria con el carácter a imprimir. A continuación se prueba el bit de estado de la impresora para

determinar si está preparada para aceptar el carácter. En tanto que no esté preparada para imprimir, se vuelve a la dirección ESPERA y se efectúa el bucle. Cuando la impresora indica que está preparada para imprimir posicionando su bit "preparada" (en este caso, por convenio, el bit 7 de la dirección ESTADO), podemos enviar el carácter. Lo transferimos del registro X al A:

```
TXA
```

y lo enviamos al registro dado de la impresora, aquí llamado IMPRD:

```
STA     IMPRD
```

Ejercicio 6.13: *Modificar el programa anterior para imprimir una cadena de* n *caracteres, en donde* n *se supondrá que es menor que 255.*

Ejercicio 6.14: *Modificar el programa anterior para imprimir una cadena de caracteres hasta que se encuentre un código de "retorno de carro".*

Compliquemos ahora el procedimiento de salida exigiendo una conversión de código y controlando varios periféricos al mismo tiempo.

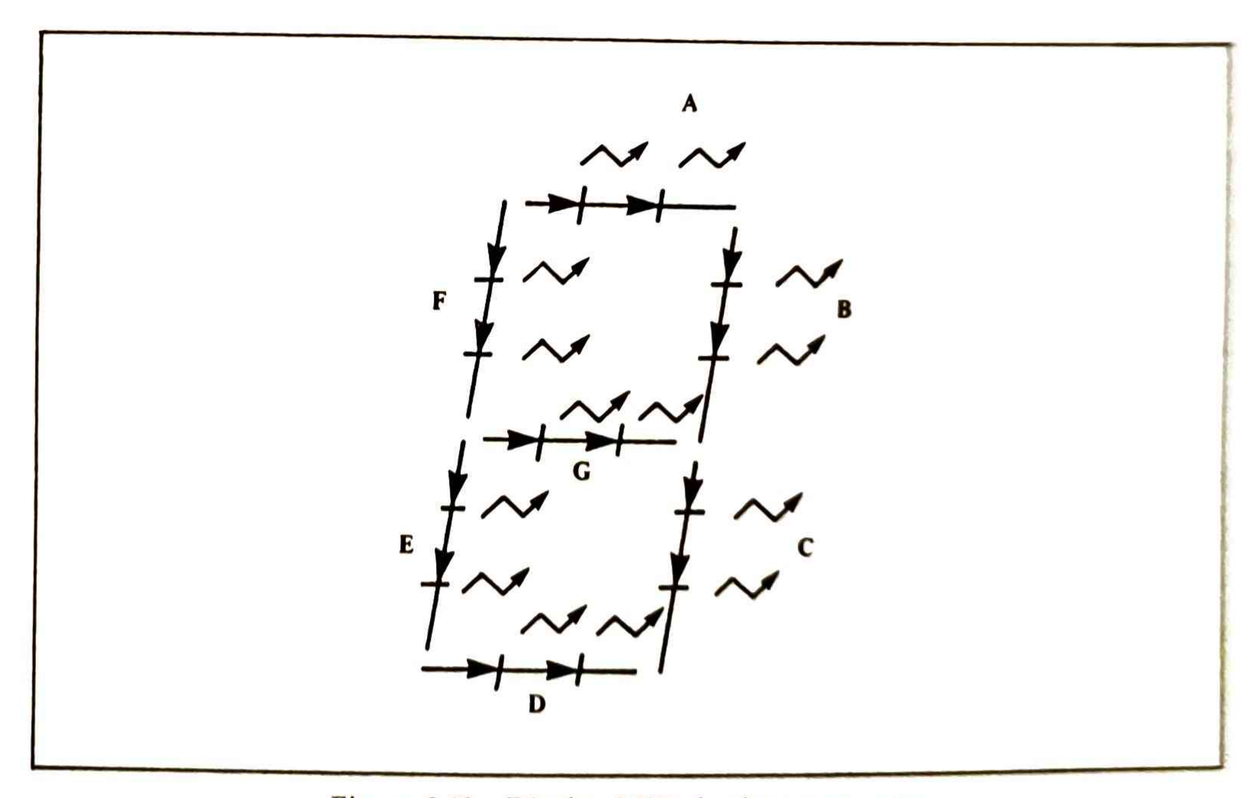


Figura 6-10 Display LED de siete segmentos.

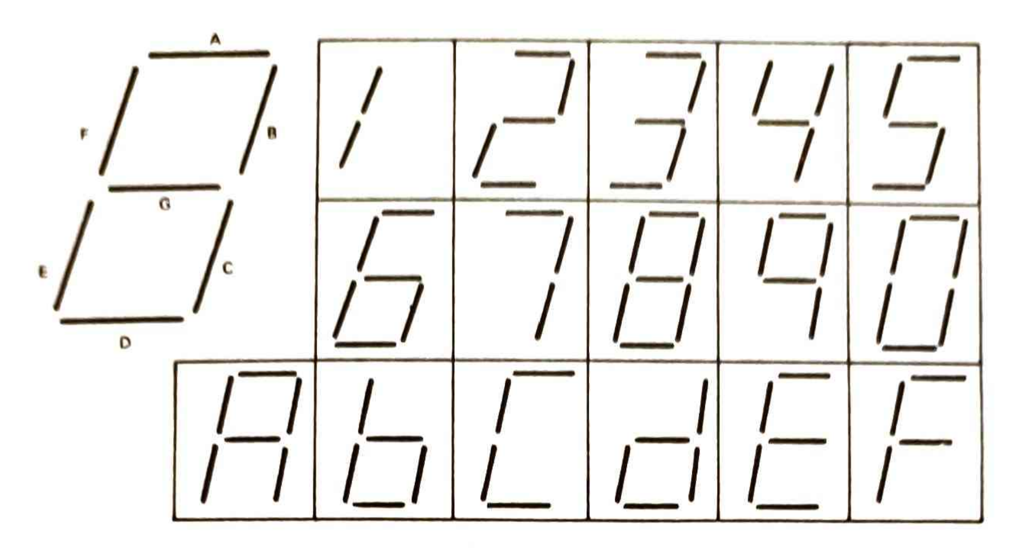


Figura 6-11 Caracteres generados con un display LED de siete segmentos.

Salida de un display de LED de 7 segmentos

Un display de diodos emisores de luz (LED) de 7 segmentos, de tipo tradicional, puede visualizar los dígitos "0" a "9", o incluso de "0" a "F" en hexadecimal, iluminando combinaciones de sus 7 segmentos. Un display de LED de 7 segmentos se muestra en la figura 6-10. Los caracteres que pueden formarse mediante este LED se ilustran en la figura 6-11. Los segmentos de un LED están etiquetados "A" a "G" como se muestra en la figura 6-10.

Por ejemplo, se visualizará "0" iluminando los segmentos "ABCDEF". Supongamos que el bit 0 de un "port" de salida está conectado al segmento "B" y así sucesivamente. El bit 7 no está utilizado. El código binario que permite iluminar "FEDCBA" (para visualizar "0") es, pues, "0111111". En hexadecimal es "3F". Haga el ejercicio siguiente.

Ejercicio 6.15: *Calcular el equivalente de 7 segmentos para los dígitos hexadecimales "0" a "F". Rellenar la tabla siguiente:*

Hex	Código LED	Hex	Código LED	Hex	Código LED	Hex	Código LED
0	3F	4		8		C	
1		5		9		D	
2		6		A		E	
3		7		B		F	

Visualicemos ahora valores hexadecimales en *diferentes* LED.

Excitación de diodos emisores de luz (LED) múltiples

Los LED no tienen memoria. Visualizan los datos sólo cuando sus líneas de segmentos están activadas. Para mantener el coste de un display de LED bastante bajo, el microprocesador visualiza la información *sucesivamente en cada LED.* La rotación entre los LED debe ser suficientemente rápida para que no haya centelleo. Ello trae consigo que el tiempo transcurrido para ir de un LED al siguiente sea inferior a 100 milisegundos. Diseñemos un programa que lo realice. Utilizaremos el registro Y para apuntar hacia el LED en el que queremos visualizar un dígito. Se supone que el acumulador contiene el valor hexadecimal a visualizar en el LED. Nuestro primer problema es convertir el valor hexadecimal en su representación de 7 segmentos. Hemos construido la tabla de equivalencia en la sección anterior. Como accedemos a una tabla, utilizaremos el direccionamiento indexado, en donde el índice de desplazamiento se proporcionará por el valor hexadecimal. Ello significa que el código de 7 segmentos para el dígito hexadecimal n.° 3 se obtendrá tomando el tercer elemento de la tabla después de la base. La dirección de la base se llamará SEGBAS. El programa correspondiente es:

```
LED       TAX                  UTILIZAR EL VALOR HEXADECI-
                               MAL COMO ÍNDICE
          LDA   SEGBAS, X      LEER EL CÓDIGO EN A
          LDX   #$00
          STX   SEGDAT         DESCONECTAR LOS EXCITADO-
                               RES DE SEGMENTOS
          STA   SEGDAT         VISUALIZAR EL DÍGITO DESEADO
          LDX   #$70           CUALQUIER NÚMERO BASTANTE
                               GRANDE
          STY   SEGADR
RETARDO   DEX
          BNE   RETARDO
          DEY                  APUNTAR HACIA LED SIGUIENTE
          BNE   SALIDA
          LDY   LEDNBR
SALIDA    RTS
```

El programa supone que el registro Y contiene el número del LED a iluminar a continuación y que el registro X contiene el dígito a visualizar.

El programa comienza por tomar el código de 7 segmentos que corresponde al valor hexadecimal contenido en el acumulador con sus dos primeras instrucciones. Las dos instrucciones siguientes cargan "00" como valor de los

segmentos a visualizar, es decir, los apagan todos. La instrucción siguiente elige el display adecuado: STY SEGADR.

Entonces se implanta un retardo por medio de un bucle de tres instrucciones antes de pasar al LED siguiente. Finalmente, el puntero de LED es objeto de decremento (podría incrementarse).

Si el puntero de LED se decrementa a "0", debe volverse a cargar con el número de LED más alto. Ello se realiza por las dos instrucciones siguientes. Se supone en este caso que se trata de un subprograma y la última instrucción es un RTS: "retorno desde subprograma".

Ejercicio 6.16: *Suponiendo que el programa anterior forma un subprograma, se observará que utiliza los registros internos X e Y internamente y modifica sus contenidos. En el supuesto de que el subprograma pueda utilizar libremente la zona de memoria de direcciones T1, T2, T3, T4 y T5, ¿puede añadir instrucciones al principio y al final de este programa que garanticen que, cuando se retorne del subprograma, el contenido de los registros X e Y será el mismo que cuando se introdujo el subprograma?*

Ejercicio 6.17: *El mismo ejercicio que antes, pero supóngase que la zona de memoria T1, etc., no está disponible para el subprograma. (Recomendación: Recuérdese que, en todo ordenador, hay un mecanismo incorporado que permite conservar informaciones en el orden cronológico.)*

Hemos resuelto los problemas de entradas-salidas habituales. Consideremos el caso de un periférico real: el teletipo.

Entrada/salida en teletipo

El teletipo es un periférico en serie. Envía y recibe palabras de información en un formato en serie. Cada carácter se codifica en un formato ASCII de 8 bits (la tabla ASCII se incluye en el apéndice E). Además, cara carácter va precedido por un bit de "comienzo" ("START") y termina en dos bits de "PARADA" ("STOP"). En el denominado interface de bucle de corriente de 20 mA, que es el más utilizado, el estado de la línea suele estar a «1». Ello sirve para indicar al procesador que la línea no está cortada. Un comienzo es una transición de "1" a "0". Indica al dispositivo receptor que siguen los bits de datos. El teletipo estándar funciona a un régimen de 10 caracteres por segundo. Acabamos de establecer que cada carácter requiere 11 bits. Ello significa que el teletipo transmite a 110 bits por segundo. Se dice que se trata de un dispositivo de 110 baudios. Vamos a escribir un programa que serializa los bits hacia el teletipo a la velocidad correcta deseada.

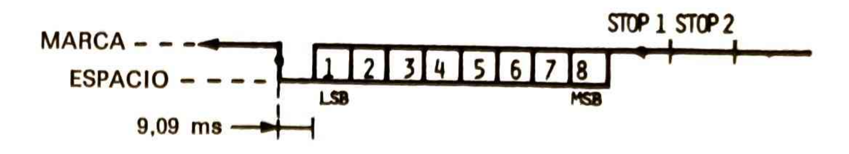


Figura 6-12 Formato de una palabra en teletipo.

El régimen de 110 bits por segundo trae consigo que los bits estén separados por 9,09 milisegundos, que tendrá que ser la duración del bucle de retardo a realizar entre los bits sucesivos. El formato de una palabra en teletipo aparece en la figura 6-12. El diagrama de flujo del programa de entrada se ilustra en la figura 6-13. El programa es el de la figura 6-14.

Obsérvese que este programa es poco diferente del diagrama de flujo de la figura 6-13. El programa debe examinarse con atención. La lógica es bastante sencilla. Lo nuevo es que, cada vez que un bit es objeto de lectura desde el teletipo (en la dirección TTYBIT), se reenvía en eco al teletipo. Esta es una característica normal del teletipo. Cada vez que un usuario pulsa una tecla, la información se transmite al procesador y luego se reenvía al mecanismo de impresión del teletipo. Cuando un carácter se imprime correctamente en el papel, ello permite verificar que las líneas de transmisión funcionan bien y que el procesador está funcionando.

Las dos primeras instrucciones forman el bucle de espera. El programa espera que el bit de estado se haga verdadero antes de comenzar a leer los bits. Como es habitual, se supone que el bit de estado está en la posición 7, pues esta posición puede probarse en una sola instrucción con BPL ("bifurcación si es más", que prueba el bit de signo).

JSR es un salto de subrutina. Utilizamos una subrutina RETARDO para realizar el retardo de 9,09 ms. Obsérvese que RETARDO puede ser un bucle programado o puede utilizarse el temporizador de hardware, si hay uno en nuestro sistema.

El primer bit que llega es el de comienzo. Debe reenviarse en eco hacia el teletipo, pero, aparte de ello, se ignora. Esto se realiza por las instrucciones 4 y 5.

De nuevo esperamos el siguiente bit. Pero esta vez se trata de un verdadero bit de datos y debemos salvaguardarlo. Puesto que todas las instrucciones de desplazamiento envían un bit al indicador de acarreo, necesitamos dos instrucciones para conservar nuestro bit de dato (el "X" de la figura 6-15): una para enviarlo al acarreo C ("LSR A") y una para conservarlo en la posición de memoria CAR (ROL).

Hay que tener presente un problema: el "ROL" destruirá el contenido de C. Si deseamos reenviar en eco el bit de datos, debe tenerse la precaución de conservarlo antes de que desaparezca en CAR. Finalmente, realicemos el

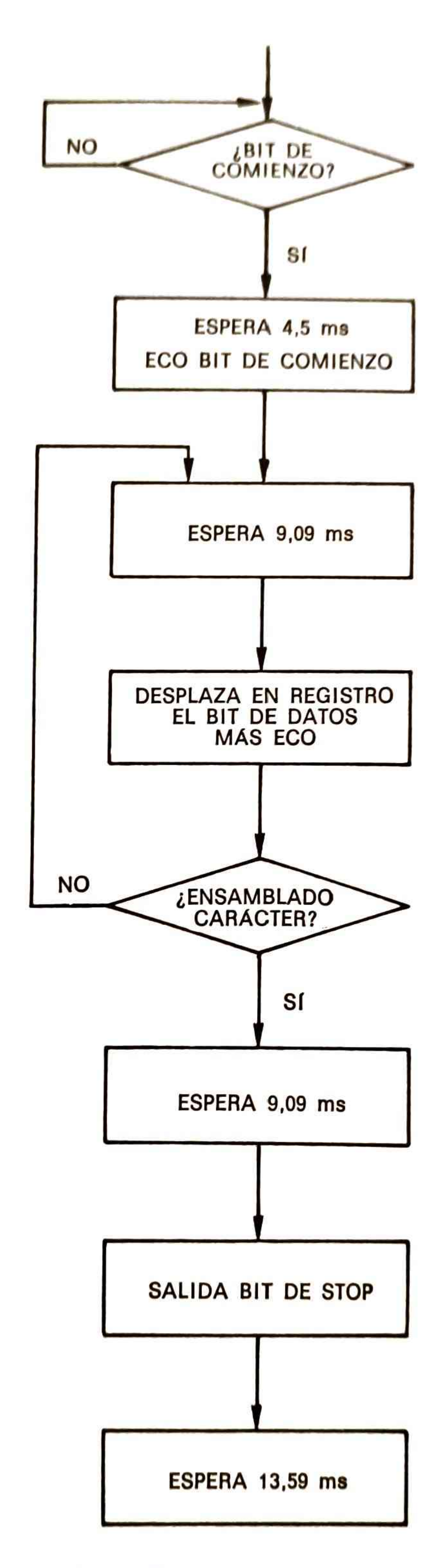


Figura 6-13 Entrada a TTY con eco.

```
TTYN       LDA   ESTADO
           BPL   TTYIN       ESCRUTINIO HABITUAL DEL ESTADO
           JSR   RETARDO     ESPERAR
           LDA   TTYBIT      BIT DE COMIENZO
           STA   TTYBIT      REENVIARLE EN ECO
           JSR   RETARDO
           LDX   #$08        CONTADOR DE BITS
SIGUIENTE  LDA   TTYBIT      CONSERVAR ENTRADA
           STA   TTYBIT      REENVIARLE EN ECO
           LSR   A           CONSERVAR BIT EN ACARREO
           ROL   CAR         CONSERVAR BIT EN CARACTER
           JSR   RETARDO
           DEX               BIT SIGUIENTE
           BNE   SIGUIENTE
           LDA   TTYBIT      BIT DE PARADA
           STA   TTYBIT
           JSR   RETARDO
           RTS
```

Figura 6-14 Entrada desde el teletipo.

eco del bit de datos (STA TTYBIT) y esperemos el siguiente (JSR RETARDO) hasta haber acumulado 8 bits de datos (DEX).

Cuando decrementemos a cero, los 8 bits están en CAR. Ya sólo nos resta reenviar los bits de PARADA (STOP) y habremos acabado.

Ejercicio 6.18: *Escribir la rutina de retardo que produzca el retardo de 9,09 milisegundos (subrutina RETARDO).*

Ejercicio 6.19: *Con la ayuda del ejemplo del programa anteriormente desarrollado, escribir un programa IMPRC que imprima en el teletipo el contenido de la posición de memoria CAR.*

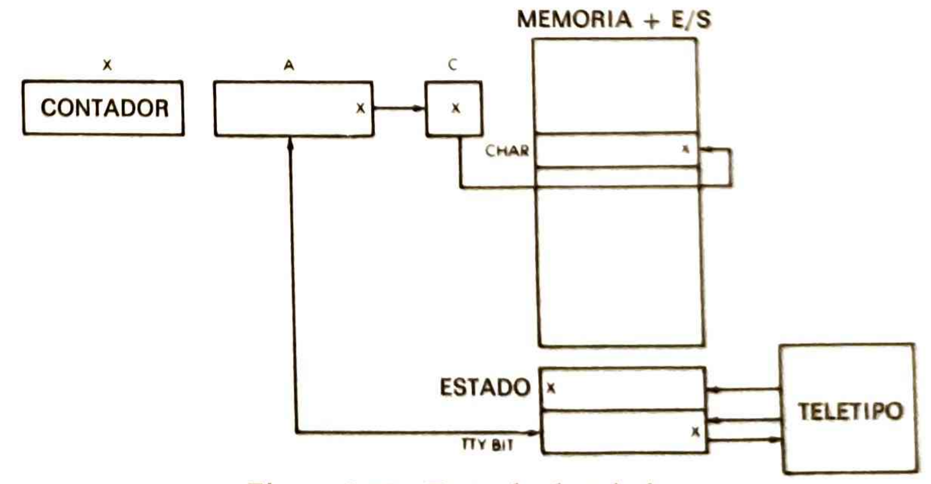


Figura 6-15 Entrada de teletipo.

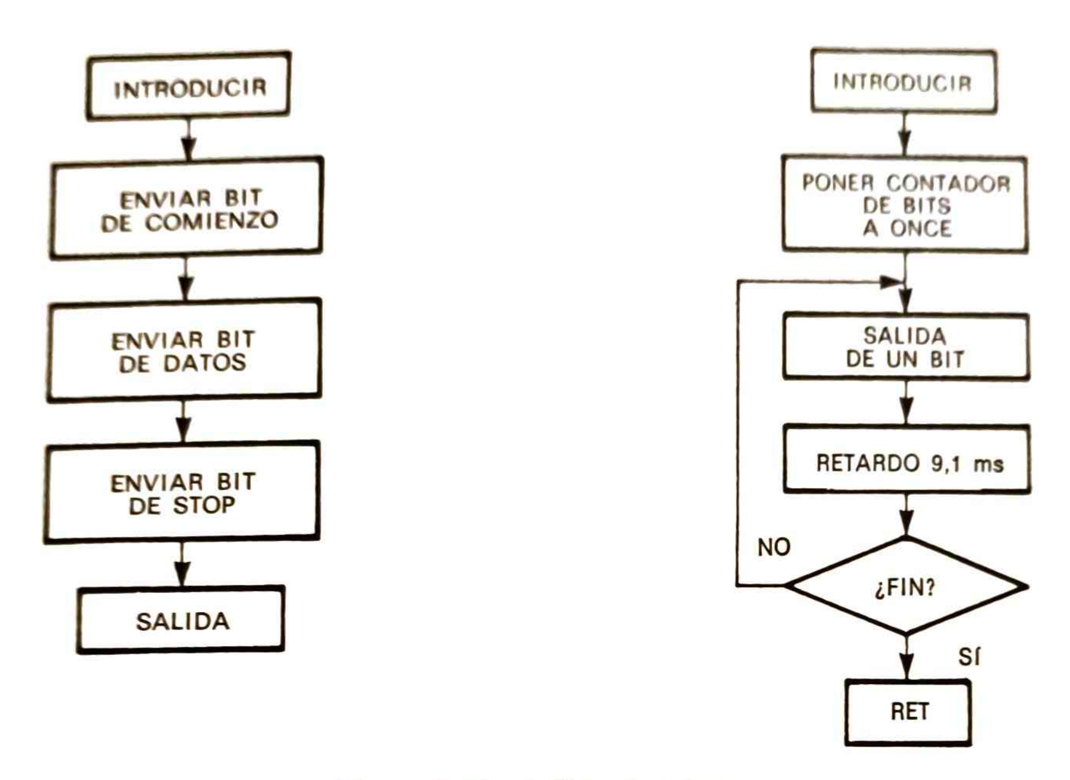


Figura 6-16 Salida de teletipo.

Ejercicio 6.20: *Modificar el programa de modo que espere un bit de comienzo (START) en lugar de un bit de estado.*

Impresión de una cadena de caracteres

Supondremos que la rutina IMPRC (ver ejercicio 6.18) se encarga de imprimir un carácter en nuestra impresora, o de visualizar en cualquier dis-

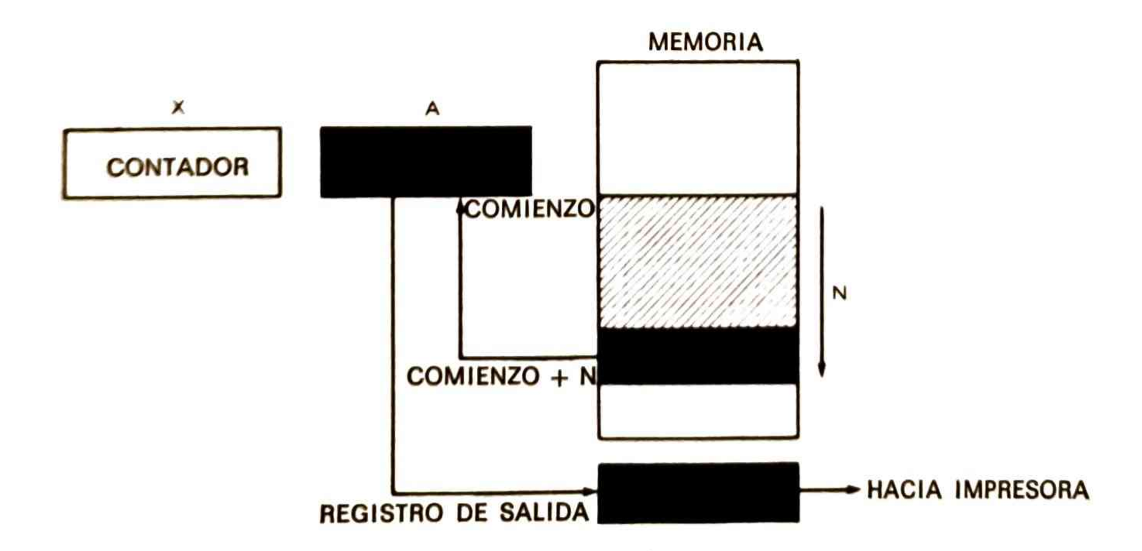


Figura 6-17 Impresión de un bloque de memoria.

positivo de salida. En este caso imprimiremos el contenido de las posiciones de memoria START + N a START (comienzo).

Naturalmente, utilizaremos el modo de direccionamiento indexado y el programa es evidente:

```
IMPCADENA   LDX   #N           NÚMERO DE PALABRAS
SIGUIENTE   LDA   START + N
            JSR   IMPRC
            DEX
            BPL   SIGUIENTE
```

RESUMEN DE LOS PERIFÉRICOS

Hemos descrito hasta ahora las técnicas básicas de programación utilizadas para comunicar con dispositivos de entrada/salida (periféricos) típicos. Además de la transferencia de datos propiamente dicha, será necesario posicionar uno o varios registros de control dentro de cada dispositivo de E/S con el fin de preparar correctamente las velocidades de transferencia, los mecanismos de interrupción y las demás opciones. Debe consultarse el manual correspondiente a cada dispositivo.

Ya hemos aprendido a trabajar con periféricos aislados. Pero, en un sistema real, todos los periféricos están conectados a los buses y pueden requerir servicio simultáneamente. ¿Cómo nos vamos a repartir el tiempo del procesador?

ORGANIZACIÓN DE ENTRADA/SALIDA

Puesto que las peticiones de entrada/salida pueden llegar simultáneamente, en cualquier sistema es preciso implantar un mecanismo de organización capaz de determinar en qué orden se atenderán las peticiones. Tres técnicas básicas de entrada/salida son utilizadas y pueden combinarse. Estas son: el escrutinio, las interrupciones y DMA. Se describirán aquí el escrutinio y las interrupciones. DMA es esencialmente una técnica de hardware y, como tal, no la describiremos en este libro.

Escrutinio

Conceptualmente, el escrutinio o sondeo ("polling") es el método más sencillo para trabajar con periféricos múltiples. En esta estrategia, el procesador interroga sucesivamente a los dispositivos conectados a los buses. Si un

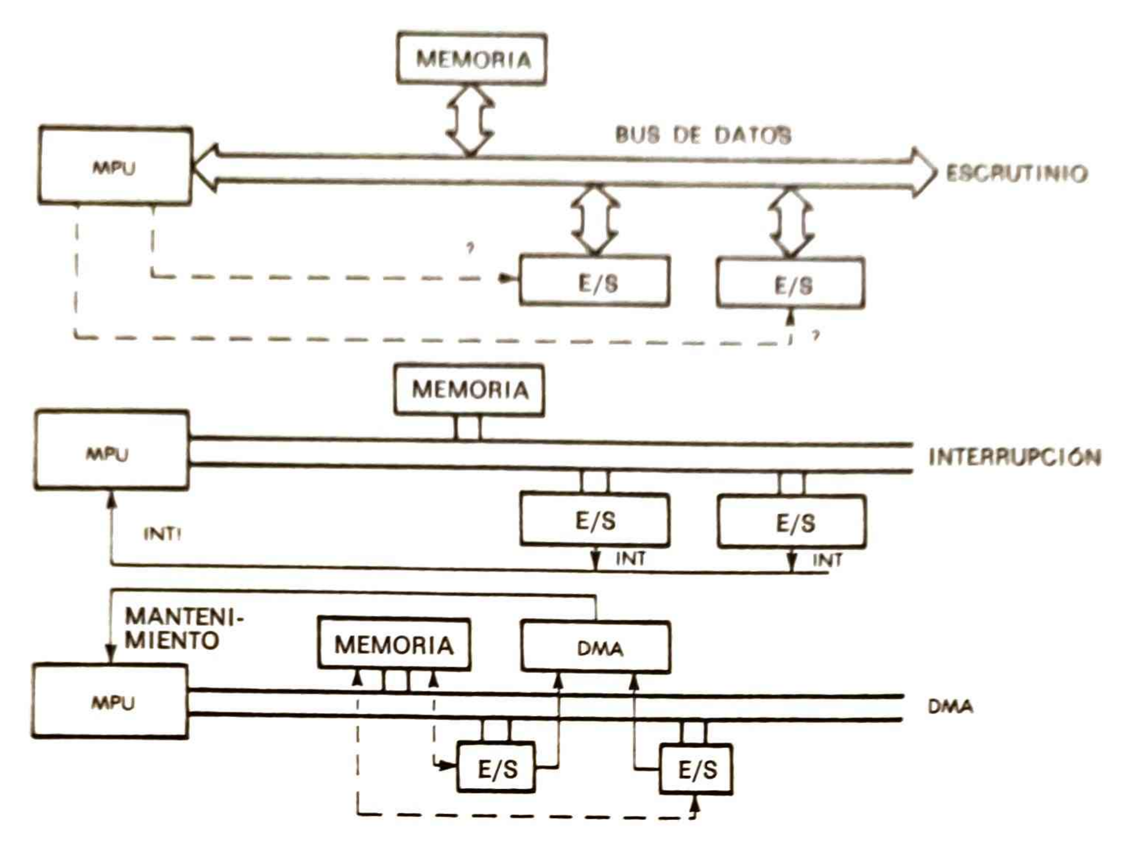


Figura 6-18 Tres métodos de control de E/S.

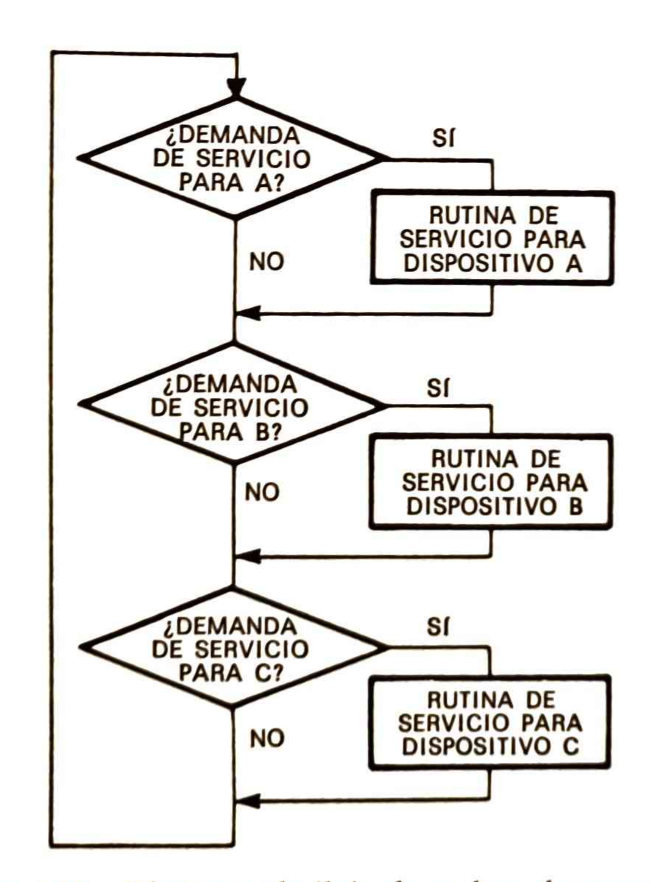


Figura 6-19 Diagrama de flujo de un lazo de escrutinio.

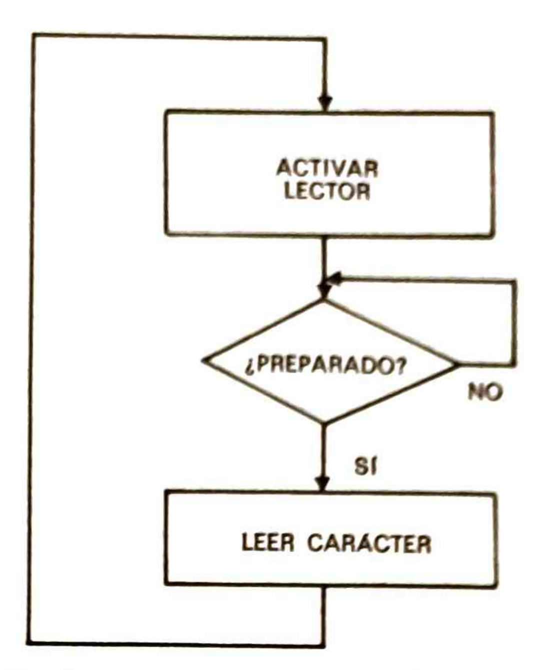


Figura 6-20 Lectura por una lectora de cinta perforada.

dispositivo requiere servicio, se le atenderá oportunamente. Si no lo requiere, se examinará el periférico siguiente. El escrutinio, o sondeo, no se utiliza simplemente para los periféricos sino también para *cualquier rutina de servicio* de dispositivos.

Por ejemplo, si el sistema está provisto de un teletipo, una grabadora de cinta y una pantalla de rayos catódicos, la rutina de escrutinio interrogará

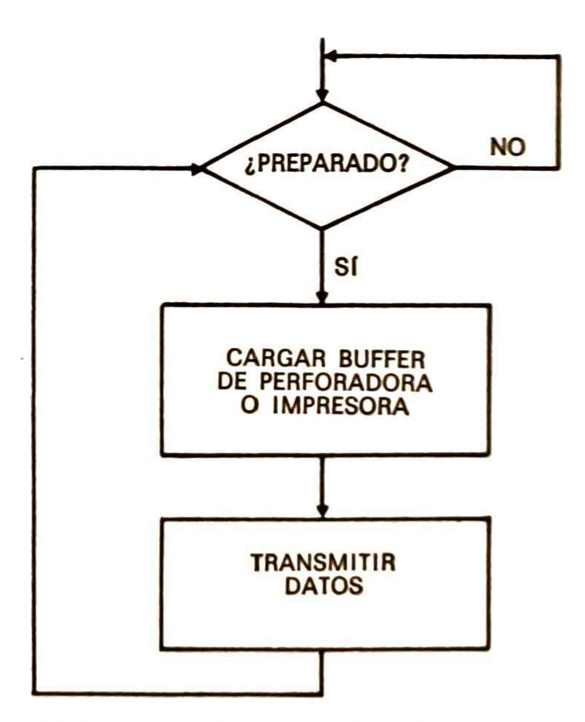


Figura 6-21 Impresión en perforadora o impresora.

primero al teletipo: "¿Tiene un carácter que transmitir?" También interrogará a la *rutina de salida* en el teletipo, preguntando: "¿Tiene un carácter que enviar?" A continuación, suponiendo que las respuestas sean negativas hasta ahora, interrogaría a las rutinas de la grabadora de cinta y finalmente a la pantalla de rayos catódicos. En el caso de que haya un solo periférico conectado al sistema, el escrutinio se utilizará también para determinar si necesita servicio. Por ejemplo, los diagramas de flujo para lectura a partir de una lectora de cinta de papel y para impresión en una impresora aparecen en las figuras 6-20 y 6-21.

Ejemplo: Un bucle de escrutinio para los periféricos 1, 2, 3, 4 (fig. 6-18).

```
ESCRUTINIO4  LDA    ESTADO1       ¿PERIFÉRICO1?
             BMI    UNO           EL BIT DE PETICIÓN
                                  DE SERVICIO ES EL 7
             LDA    ESTADO2       ¿PERIFÉRICO2?
             BMI    DOS
             LDA    ESTADO3       ¿PERIFÉRICO3?
             BMI    TRES
             LDA    ESTADO4       ¿PERIFÉRICO4?
             BMI    CUATRO
             JMP    ESCRUTINIO4   PROBAR DE NUEVO
```

El bit 7 del registro de estado para cada periférico es "1", cuando hay necesidad de servicio. Cuando se detecta una petición, este programa se bifurca hacia la rutina del periférico en la dirección UNO para el periférico 1, DOS para el periférico 2, etc.

Las ventajas del escrutinio son evidentes: es simple, no requiere ningún hardware y conserva la sincronización de todas las entradas/salidas con el funcionamiento del programa. Su desventaja es también evidente: la mayor parte del tiempo del procesador se desperdicia interrogando periféricos que no tenían necesidad de servicio. Además, el procesador corre el riesgo de ocuparse demasiado tarde de un periférico, perdiendo mucho tiempo.

Es, pues, deseable disponer de otro mecanismo que garantice poder utilizar el tiempo del procesador en efectuar cálculos útiles, y no en interrogar todo el tiempo a periféricos que no tienen necesidad de servicio. No obstante, insistimos en el hecho de que el escrutino es universalmente utilizado siempre que el microprocesador no tenga nada mejor que hacer, dado que mantiene sencilla la organización global. Examinemos ahora la alternativa esencial del escrutinio: las interrupciones.

Interrupciones

El concepto de interrupciones se ilustra en la figura 6-18. Se dispone de una línea de hardware especial, la línea de interrupción, que está conectada a un terminal especial del microprocesador. Dispositivos de entrada/salida múltiples pueden conectarse a esta línea de interrupción. Cuando cualquiera de ellos precisa servicio, envía un nivel o un impulso en esta línea. La señal de interrupción es la petición de servicio que emana de un dispositivo de entrada/salida hacia el procesador. Examinemos, la respuesta del procesador a esta interrupción.

En cualquier caso, el procesador termina la instrucción que estaba en curso de ejecución pues, de no ser así, produciría un caos en el interior del microprocesador. A continuación, el microprocesador debe bifurcarse a una rutina de tratamiento de la interrupción que procesará la interrupción. La bifurcación a una subrutina de tal naturaleza trae consigo que el contenido del contador de programa deba salvaguardarse en la pila. *Una interrupción debe, pues, dar lugar a la salvaguarda automática del contador de programa en la pila.* Además, el registro de estado (P) debe salvaguardarse también automáticamente, puesto que su contenido se alterará por cualquier instrucción posterior. Finalmente, si la rutina de manipulación de la interrupción debe modificar registros internos, éstos deben conservarse también en la pila.

Una vez que se hayan conservado todos estos registros, se puede bifurcar a la dirección adecuada de la rutina de interrupción. Al final de esta rutina, todos los registros deben restaurarse y se ha de ejecutar una instrucción especial de retorno de interrupción con el fin de reanudar la ejecución del programa principal. Examinemos con más detalle las dos líneas de interrupción del 6502.

Interrupciones del 6502

El 6502 tiene dos líneas de interrupción, IRQ y NMI. IRQ es la línea de interrupción normal, mientras que NMI es una interrupción más prioritaria y no enmascarable. Examinemos su funcionamiento.

IRQ es una interrupción activada por nivel o flanco. El estado de la línea IRQ se probará, o ignorará, por el microprocesador según el valor del indicador interno I (indicador de máscara de interrupción). Supondremos, primero, que las interrupciones están autorizadas. Cada vez que se activa IRQ, la interrupción se detectará por el microprocesador. Tan pronto como se acepte la interrupción (a la terminación de la instrucción en curso), el indicador I se pone automáticamente a "1". Ello impedirá que el microprocesador vuelva a interrumpirse en el momento en que se manipulan los registros internos. A continuación, el 6502 conserva automáticamente el contenido

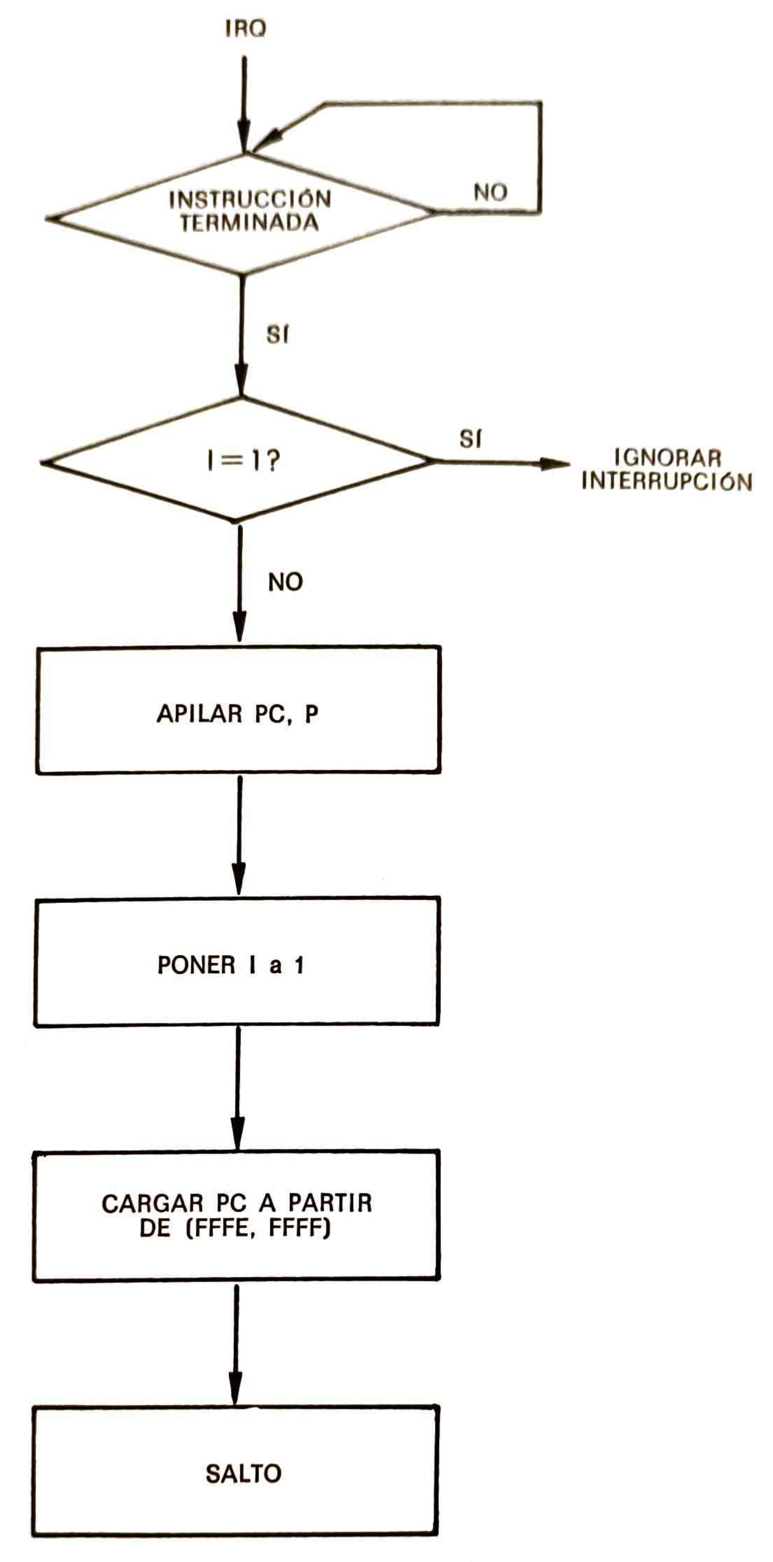


Figura 6-22 Tratamiento de interrupciones.

de PC (contador de programa) y el de P (registro de estado) en la pila. El aspecto de la pila, después de una interrupción, se ilustra por la figura **6-23**.

A continuación, el 6502 buscará automáticamente el contenido de las posiciones de memoria "FFFE" y "FFFF". Esta posición de memoria de

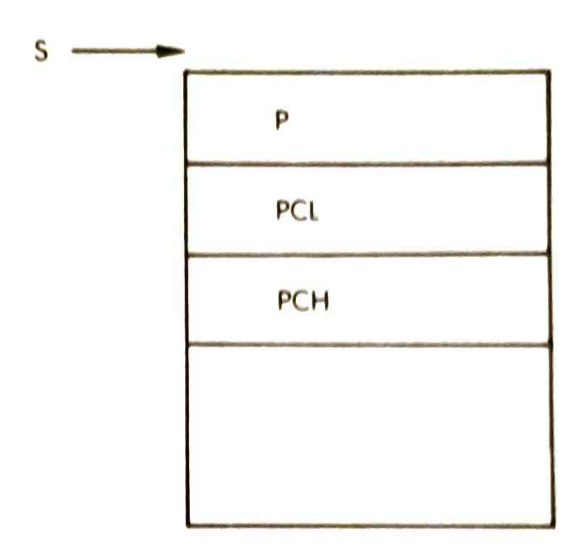


Figura 6-23 La pila del 6502 después de una interrupción.

16 bits contiene el *vector de interrupción*. El 6502 buscará y cargará el contenido de esta dirección y luego, se bifurca al vector de 16 bits especificado. El usuario tiene la responsabilidad de depositar esta dirección de vectorización en "FFFE" y "FFFF". Sin embargo, varios dispositivos pueden conectarse a la línea IRQ. En este caso, efectuaremos una bifurcación a una sola rutina de manipulación de interrupción. ¿Cómo vamos a hacer la distinción entre los diferentes periféricos? Esto se estudiará en la sección siguiente.

La interrupción NMI es prácticamente idéntica a IRQ, salvo que no puede enmascararse por el bit I. Se trata de una interrupción prioritaria que suele utilizarse para las averías de la alimentación eléctrica. Aparte de ello, el funcionamiento es idéntico, con la salvedad de que el procesador se bifurca automáticamente al contenido de "FFFA"-"FFFB". Esto se ilustra en la figura 6-24.

El retorno desde una interrupción se realiza mediante la instrucción RTI. Esta instrucción recupera, en el microprocesador, las tres palabras de la parte alta (cima) de la pila, que contiene P y PC (el contador de programa de 16 bits). El programa que haya sido interrumpido, puede reanudarse entonces. El estado interno de la máquina es exactamente idéntico al que existía cuando se produjo la interrupción. El efecto ha sido introducir un retardo en la ejecución del programa.

Antes de volver desde una interrupción, el programador tiene la responsabilidad de liberar la interrupción que acaba de tratarse y de restaurar el indicador de inhibición de las interrupciones. Además, si la rutina de interrupción modifica el contenido de cualquier registro, tal como X o Y, el programador es específicamente responsable de la conservación de estos registros en la pila, antes de ejecutar la rutina de manipulación de la interrupción, o tratamiento propiamente dicho de la misma. De no ser así, se modificará

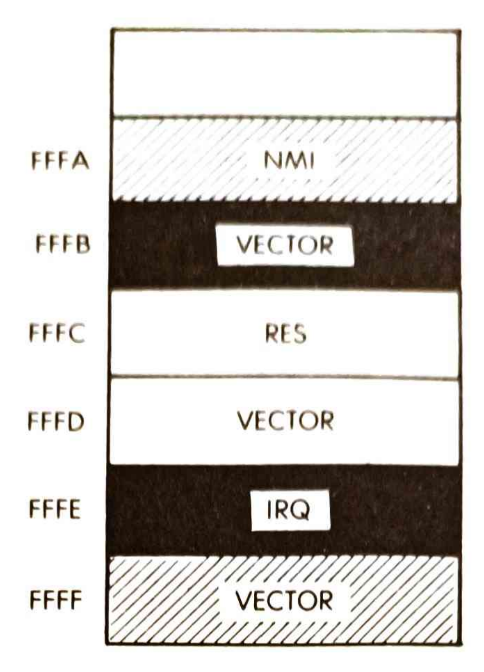


Figura 6-24 Vectores de interrupción.

el contenido de estos registros y cuando se reanude la ejecución del programa interrumpido, no será correcto.

Si se supone que la rutina de interrupción utiliza los registros A, X e Y, serán necesarias cinco instrucciones en la rutina de manipulación de interrupción para conservar estos registros. Dichas interrupciones son:

```
SAVAXY  PHA   INTRODUCIR EN LA PILA
        TXA   TRANSFERIR X EN A
        PHA   INTRODUCIRLE EN PILA
        TYA   TRANSFERIR Y EN A
        PHA   INTRODUCIRLE EN PILA
```

Lamentablemente, el 6502 no puede introducir directamente en pila el contenido de A o de P. En consecuencia, la conservación de X y de Y consume tiempo y requiere 4 instrucciones. Esto se ilustra en la figura 6-25.

A la terminación de la rutina de interrupción, es preciso restaurar estos registros y dicha rutina debe concluirse con la secuencia de seis instrucciones:

```
PLA   EXTRAER Y DE LA PILA
TAY   RESTAURAR Y
PLA   EXTRAER X DE LA PILA
TAX   RESTAURAR X
PLA   RESTAURAR A
RTI   SALIDA DE INTERRUPCIÓN
```

Ejercicio 6.21: *Con el empleo de la tabla del apéndice D que indica el número de ciclos por instrucción, calcular cuánto tiempo se perderá al efectuar la salvaguarda y luego restaurar los registros A, X e Y.*

Para una comparación gráfica entre el proceso de escrutinio y el de interrupción, es preciso referirse a la figura **6-18**, en donde el escrutinio se ilustra en la parte superior y el proceso de interrupción debajo. Se puede observar que, en la técnica de escrutinio, el programa desperdicia mucho tiempo en la espera. Con el empleo de interrupciones, el programa se interrumpe, la interrupción es objeto de tratamiento y luego se reanuda el programa. No obstante, la desventaja evidente de una interrupción es introducir varias instrucciones suplementarias al principio y al final, lo que da lugar a un retardo antes de que pueda ejecutarse la primera instrucción de la rutina de manipulación del periférico. Se trata de un tiempo perdido suplementario.

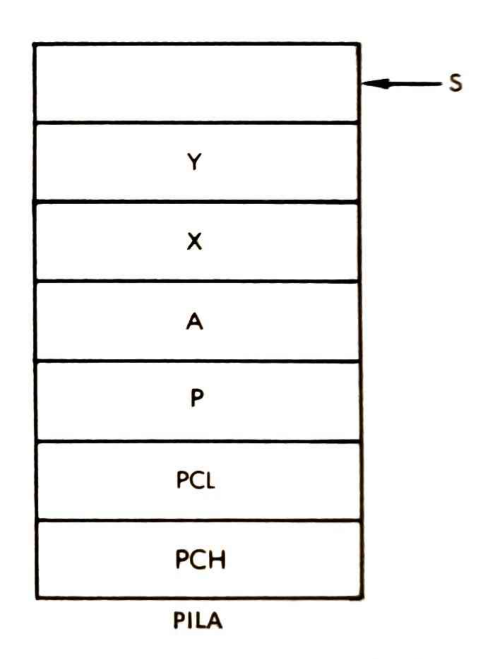


Figura 6-25 Conservación de todos los registros.

Habiendo aclarado el funcionamiento de las dos líneas de interrupciones, consideramos ahora los dos problemas importantes que restan:

1. ¿Cómo resolver el problema de varios periféricos que disparan (activan) una interrupción al mismo tiempo?
2. ¿Cómo resolver el problema de la llegada de una interrupción mientras está siendo objeto de tratamiento otra interrupción?

Periféricos múltiples conectados a una misma línea de interrupción

Cada vez que se produce una interrupción, el procesador se bifurca automáticamente a una dirección contenida en "FFFE-FFFF" (para una IRQ) o en "FFFA-FFFB" (para NMI). Antes de poder hacer un tratamiento efectivo cualquiera, la rutina de interrupción debe determinar cuál es el periférico que ha "disparado" la interrupción. Como es habitual, se dispone de dos métodos para identificar el periférico: un método de software y un método de hardware.

En el método de software se utiliza la técnica del escrutinio. El microprocesador interroga sucesivamente cada periférico y le hace la pregunta "¿Disparó la interrupción?". De no ser así, interroga al siguiente. En la figura 6-26 se ilustra este proceso. Un programa tipo es:

```
LDA   ESTADO 1
BMI   UNO
LDA   ESTADO 2
BMI   DOS
```

El método de hardware utiliza componentes suplementarios pero proporciona la dirección del periférico que activa la interrupción, al mismo tiempo que la petición de interrupción. El circuito utilizado ahora, de forma universal, para esta aplicación se denomina PIC (priority interrupt controller-controlador de prioridad de interrupciones). Dicho circuito PIC colocará automáticamente en el bus de datos la dirección de bifurcación exacta deseada para el periférico objeto de interrupción. Cuando el 6502 pasa a FFFE-

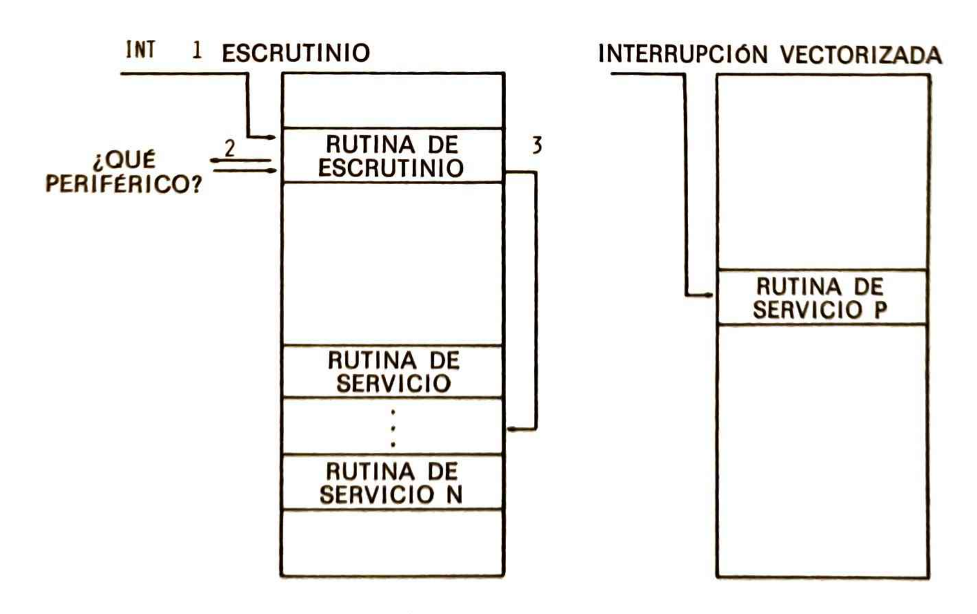


Figura 6-26 Relación entre interrupción vectorizada y de escrutinio.

FFFF, buscará esta dirección de vectorización. El concepto se ilustra en la figura 6-26.

En la mayor parte de los casos, la velocidad de respuesta a una interrupción no es crucial y se utiliza el escrutinio. Si el tiempo de respuesta es de gran importancia, es preciso emplear la solución de hardware.

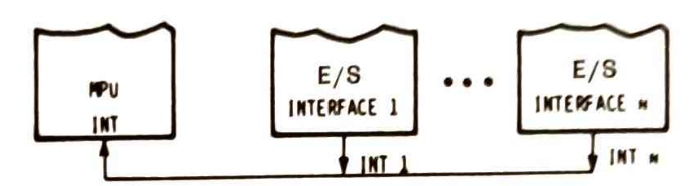


Figura 6-27 Diferentes periféricos pueden utilizar la misma línea de interrupción.

Interrupciones simultáneas

El problema siguiente que puede plantearse es que puede dispararse una nueva interrupción en el curso de la ejecución de una rutina de interrupción anterior. Examinemos lo que llega y cómo se utiliza la pila para resolver el problema. En el capítulo 2 hemos indicado que esto era otra utilización esencial de la pila y el tiempo se ha encargado de demostrar su uso. Nos referiremos a la figura 6-28 para ilustrar las interrupciones simultáneas. El tiempo transcurre de izquierda a derecha en la figura. El contenido de la pila aparece en la parte inferior de la ilustración. Al mirar a la izquierda, en el instante T0 el programa está en curso de ejecución. Si se desplaza hacia la derecha, en el instante T1 se produce la interrupción I1. Supondremos que

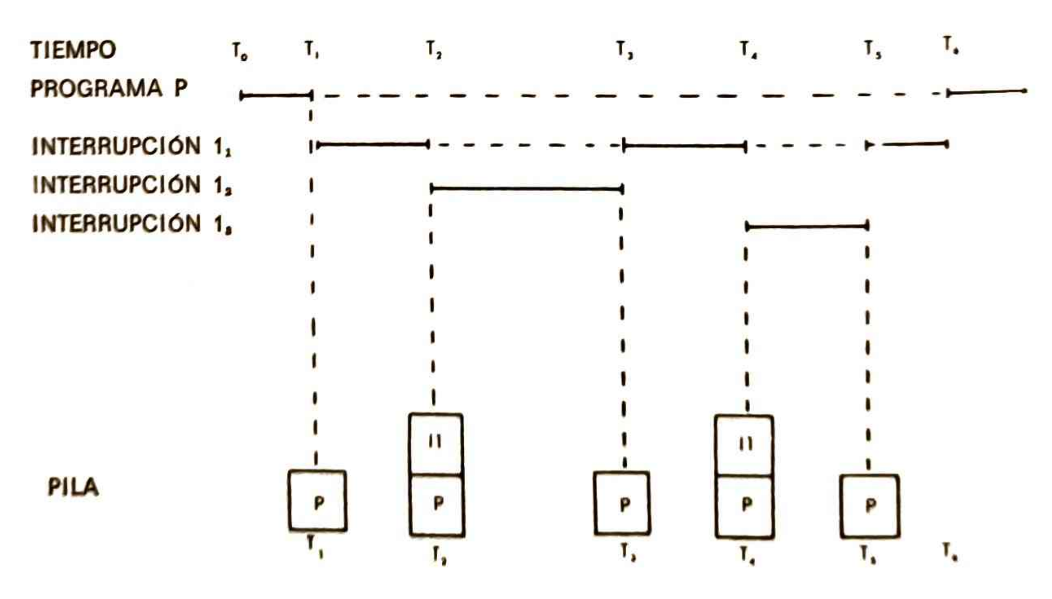


Figura 6-28 La pila durante las interrupciones.

la máscara de interrupción estaba posicionada de forma que autorizara I1. Se suspenderá el programa P. Ello se muestra en la parte inferior de la ilustración. La pila contendrá al menos el contador de programa P más cualquier registro eventual que pudiera salvaguardarse por la rutina de interrupción o I1 por sí misma.

En el instante T1, la interrupción I1 comienza a ejecutarse y sigue su ejecución hasta el instante T2. En este instante, se produce la interrupción I2. Supondremos que I2 se considera como más prioritaria que I1. Si tuviera una prioridad más baja, se hubiese ignorado hasta el final de I1. En el instante T2, los registros del contexto de I1 son introducidos en la pila y esto aparece en la parte inferior de la figura. De nuevo se introduce en la pila el contenido del contador de programa y de P. Además, la rutina de tratamiento de I2 podría decidir la conservación de algunos registros suplementarios. I2 se ejecutará ahora hasta la terminación en el instante T3.

Cuando termina I2, el contenido de la pila se reenvía automáticamente al 6502 y ello aparece en la parte inferior de la figura 6-28. Así, la interrupción I1 reanuda su ejecución. Lamentablemente, en el instante T4 vuelve a producirse una interrupción más prioritaria I3. Podemos ver, en la parte inferior de la ilustración, que los registros para I1 se introducen de nuevo en la pila. La interrupción I3 se ejecuta de T4 a T5 y se termina en T5. En este instante se extrae el contenido de la cima de la pila y se lleva al 6502 y la interrupción I1 reanuda la ejecución. Esta vez llega al final y se termina en T6. Aquí, los registros que quedaban en la pila se reenvían al 6502 y el programa P puede reanudar su ejecución. El lector verificará que, en este instante, al pila está vacía. De hecho, el número de líneas de trazos, que indica suspensiones de programa, señala al mismo tiempo el número de niveles que existen en la pila en este instante.

Ejercicio 6.22: *Si suponemos que cada vez que se produce una interrupción el contador de programa PC, el registro P y el acumulador se salvaguardarán, ello ocupará un mínimo de 4 bytes. (En la práctica X e Y pueden salvaguardarse también, lo que requeriría 6 bytes.) Suponiendo, pues, que en la pila sólo se salvaguardan o conservan tres registros ¿cuántos niveles de interrupción permite el 6502? (recuérdese que la pila está limitada a 256 bytes en la página 1).*

Ejercicio 6.23: *Suponiendo esta vez que se conservan 5 registros en la pila ¿cuál es el número máximo de interrupciones simultáneas que se pueden tratar? ¿Hay otros factores que contribuyen a disminuir todavía más el número de interrupciones simultáneas?*

No obstante, es preciso insistir en el hecho de que, en la práctica, los sistemas de microprocesador suelen estar conectados a un pequeño número de periféricos que utilizan las interrupciones. Es, pues, poco probable que se produzca un gran número de interrupciones simultáneas en un sistema de tal naturaleza.

Hasta ahora hemos resuelto todos los problemas que suelen estar asociados con las interrupciones. De hecho, su utilización es simple e incluso el programador principiante podría obtener ventajas. Completemos nuestro análisis de los recursos del 6502 introduciendo una instrucción suplementaria, cuyo efecto es idéntico al de una interrupción síncrona.

Ruptura (BRK)

La instrucción BRK del 6502 es equivalente a una interrupción de software. Puede insertarse en un programa y, exactamente como en el caso de una IRQ, da lugar a la conservación automática de PC y P y luego a una bifurcación indirecta a FFFE-FFFF. Esta instrucción puede utilizarse ventajosamente para crear interrupciones programadas, durante la puesta a punto, o depuración, de un programa. Esto dará lugar a la creación de un punto de ruptura, a la interrupción del programa en un lugar predeterminado y a la bifurcación a una rutina que, normalmente, permitirá al usuario analizar el programa. Como el efecto global de BRK y de una interrupción es el mismo después de que hayan tenido lugar, es preciso proporcionar un medio de determinar si era una interrupción o una BRK. El 6502 pone a "1" un indicador B del registro P (conservado en la pila), si era una ruptura y a "0" si era una interrupción. Se puede probar el estado de este bit con la ayuda del programa simple siguiente:

```
TEST B   PLA          LEER LA CIMA DE LA PILA EN A
         PHA          VOLVER A ESCRIBIRLA
         AND #$10     ENMASCARAR EL BIT B
         BNE PRBRK    PASAR AL PROGRAMA DE RUPTURA
```

Este programa de prueba suele insertarse al final de la secuencia de escrutinio que determina el periférico que había activado la interrupción.

Observación: Una peculiaridad de BRK es conservar automáticamente el contenido del contador de programa *más* 2. Puesto que la instrucción BRK no ocupa más que 1 byte, el programador puede tener, a veces, que ajustar el contenido del contador de programa en la pila con la ayuda de una instrucción de incremento, o de decremento, con el fin de reanudar la ejecución en la dirección correcta. En particular, BRK se utiliza de forma intensiva durante la depuración ("debugging") escribiéndola en el lugar de otra instruc-

ción del programa. Si el programa se vuelve a ensamblar antes de la ejecución, normalmente será preciso decrementar en "1" el contenido del contador de programa que haya sido conservado.

RESUMEN

Hemos presentado en este capítulo el conjunto de técnicas utilizadas para comunicar con el mundo exterior. Desde las rutinas elementales de entrada/salida hasta los programas más complejos para comunicar con los periféricos existentes, hemos aprendido a desarrollar todos los programas habituales e incluso hemos examinado la eficacia de programas de referencia de prestaciones ("benchmark") en el caso de una transferencia en paralelo y en el de una conversión serie-paralelo. Finalmente, hemos aprendido a organizar el funcionamiento de periféricos múltiples mediante escrutinio o mediante interrupciones. Naturalmente, muchos otros periféricos de todas clases podrían conectarse a un sistema. Con el conjunto de las técnicas que se han presentado hasta ahora y con una comprensión de los periféricos correspondientes, debe ser posible resolver la mayor parte de los problemas habituales.

En el capítulo siguiente, examinaremos las características reales de las pastillas integradas de interface de E/S normalmente conectadas al 6502. A continuación consideraremos las estructuras de datos fundamentales que el programador puede considerar dignas de utilización.

EJERCICIOS

Ejercicio 6.24: *Un display de LED de 7 segmentos puede visualizar también caracteres distintos de los del alfabeto hexadecimal. Calcular los códigos para: H, I, J, L, O, P, S, U, Y, g, h, i, j, l, n, o, p, r, t, u, y.*

Ejercicio 6.25: *El diagrama de flujo del tratamiento de las interrupciones aparece en la figura 6-29 adjunta. Dar respuesta a las preguntas siguientes:*

a) ¿Qué es lo que se realiza por hardware? ¿Qué es lo que se hace mediante software?
b) ¿Cuál es el uso de la máscara?
c) ¿Cuántos registros deben conservarse?
d) ¿Cómo se identifica el dispositivo de interrupción?
e) ¿Qué hace la instrucción de RTI? ¿En qué se diferencia de un retorno de subrutina?

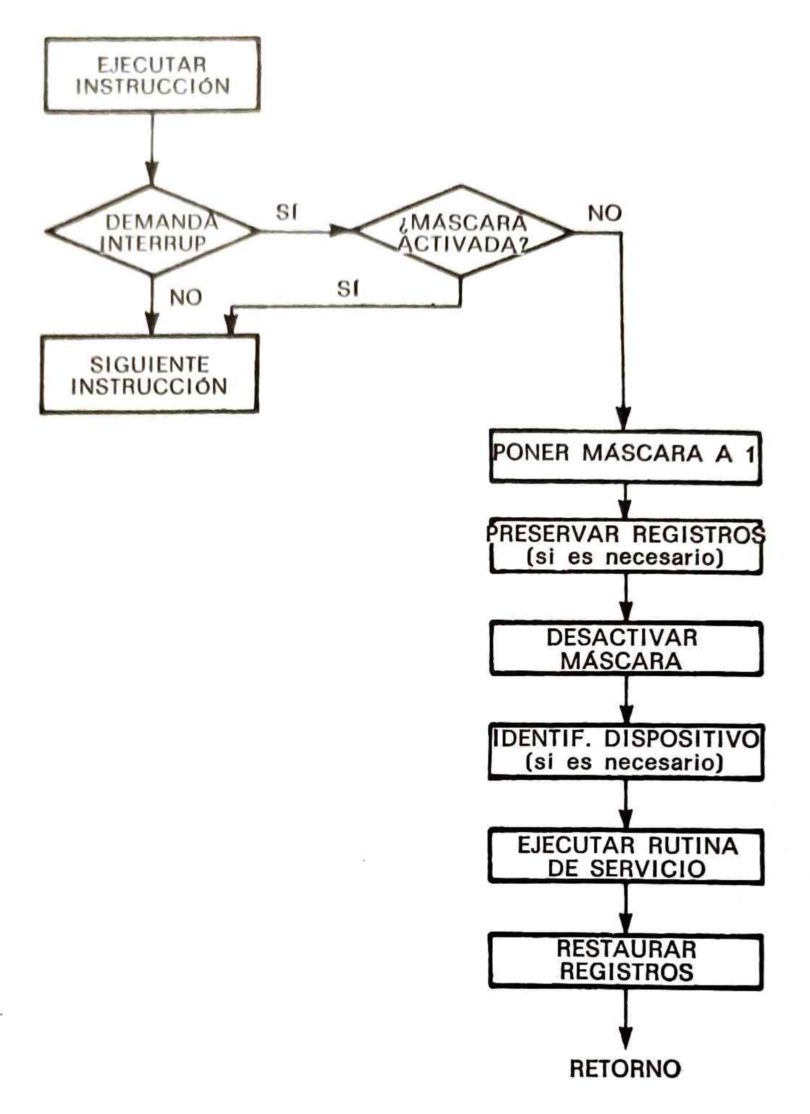


Figura 6-29 Lógica de las interrupciones.

f) Indíquese una forma de tratamiento de una situación de desbordamiento de pila.

g) ¿Cuál es el "tiempo perdido" introducido por el mecanismo de interrupción?

7 Dispositivos de entrada/salida

INTRODUCCIÓN

Hemos aprendido a programar el microprocesador 6502 en las situaciones más habituales. Sin embargo es preciso hacer una mención particular de las pastillas integradas ("chips") de entrada/salida normalmente conectadas al microprocesador. Gracias al progreso en la integración a gran escala (LSI), se han introducido nuevas pastillas que no existían antes. En consecuencia, la programación de un sistema requiere, naturalmente, programar primero el propio microprocesador, pero también *programar las pastillas de entrada/salida*. De hecho, suele ser más difícil recordar cómo programar las diferentes opciones de control de una pastilla de entrada/salida que programar el propio microprocesador. Y ello no se debe a que sea más difícil la programación en sí misma, sino porque cada uno de estos dispositivos tiene sus propias peculiaridades. En este caso, vamos a examinar primero el dispositivo de entrada/salida más general, la pastilla integrada de entrada/salida programable (PIO-programmable input/output) y luego introduciremos las "mejoras" en este PIO que suelen utilizarse con el 6502 que se denominan: 6520, 6530, 6522 y 6532.

El PIO estándar (6520)

No hay ningún PIO estándar. Sin embargo, el 6520 es prácticamente análogo, desde el punto de vista funcional, a todos los PIO proporcionados por otros fabricantes para la misma finalidad. El objeto de un PIO es proporcionar una conexión "multiport" para dispositivos de entrada/salida (un "port" es simplemente un conjunto de 8 líneas de entrada/salida). Cada PIO

proporciona, como mínimo, dos conjuntos de líneas de entrada/salida de 8 bits. Cada dispositivo de entrada/salida (E/S) necesita un *buffer de datos* para estabilizar a la salida el contenido del bus de datos. Nuestro PIO tendrá, pues, como mínimo, un buffer para cada "port".

Además hemos establecido que el microordenador utilizará un procedimiento de sincronización de los intercambios o *diálogo* ("handshaking") o bien *interrupciones* para comunicar con el dispositivo de E/S. El PIO utilizará también un procedimiento similar para la comunicación con el periférico. Por consiguiente, cada PIO debe estar provisto de, como mínimo, *dos líneas de control por "port"* para implantar la función de diálogo.

El microprocesador necesitará también ser capaz de leer el estado de cada "port". Cada "port" debe estar provisto de uno o más *bits de estado*. Finalmente, cada PIO ofrece un cierto número de opciones para configurar sus recursos. El programador debe ser capaz de acceder a un registro particular dentro del PIO para especificar las opciones de programación. Este es el *registro de control*. En el caso del 6520, la información del estado forma parte del registro de control.

Una posibilidad esencial del PIO es el hecho de que cada línea puede configurarse como una entrada o como una salida. El diagrama de un PIO aparece en la ilustración de la figura 7-1. El programador puede especificar si cualquier línea será una entrada o una salida. Para programar la dirección de las líneas se proporciona un *registro de dirección de los datos* para cada port. Un "0" en una posición de bit del registro de dirección de los datos especifica una entrada. Un "1" especifica una salida.

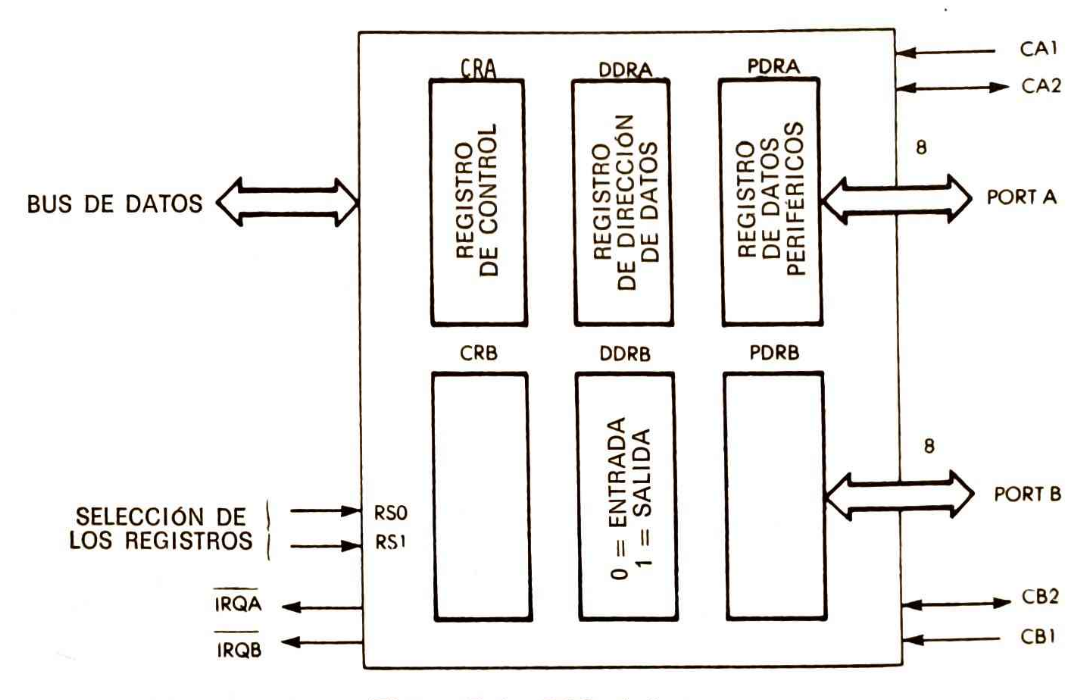


Figura 7-1 PIO típico.

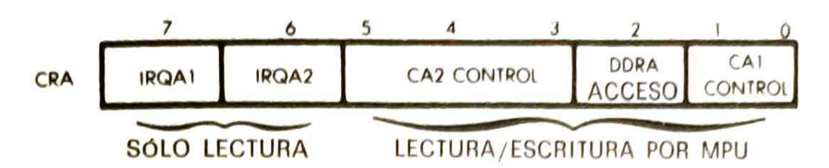


Figura 7-2 Formato de palabra de control de un PIO.

Puede ser sorprendente que "0" sea utilizado para entrada y "1" para salida, cuando realmente "0" debe corresponder a salida y "1" a entrada. Ello es deliberado: siempre que se aplique la alimentación al sistema, es de gran importancia que todas las líneas de E/S estén configuradas como *entrada*. De no ser así, si el microordenador está conectado a un periférico peligroso, podría activarlo por accidente. Cuando se efectúa un "reset", todos los registros están normalmente puestos a cero y ello da lugar a que se configuren todas las líneas periféricas del PIO como entradas. La conexión al microprocesador aparece a la izquierda de la ilustración. Naturalmente, el PIO se conecta a los 8 bits del bus de datos, al bus de dirección del microprocesador y a su bus de control. El programador especificará simplemente la dirección de cualquier registro al que desea acceder dentro del PIO. El 6520, que es compatible con el 6820 de Motorola, ha "heredado" una de sus peculiaridades: está provisto de 6 registros internos. Pero no se puede especificar más que un registro de entre cuatro. La forma en que se resuelve este problema es conmutar el valor del bit 2 del registro de control. Cuando este bit es un "0", puede seleccionar el correspondiente registro de dirección de los datos. Cuando es un "1", puede seleccionarse el registro de datos. Por consiguiente, cuando el programador quiere escribir directamente datos en el registro de dirección de datos, debe cerciorarse primero de que el bit 2 del registro de control adecuado sea cero, antes de que pueda seleccionar este registro. Esto es algo penoso para programar, pero es importante recordarlo para evitar graves dificultades.

Para averiguar el efecto de la selección de direcciones en el 6520 se consultará la tabla de la figura 7-3. RS0 y RS1 son dos señales de selección de registro que se derivan del bus de dirección. Dicho de otro modo, representan dos bits de la dirección especificada por el programador. CRA es el registro de control para port A. CRA (2) es el bit 2 de este registro. CRB es el registro de control para "port" B.

El registro de control interno

El registro de control del 6520 especifica, como hemos visto, por medio de su bit 2, un modo de selección para los registros internos del "port". Además, proporciona varias opciones para generar o detectar interrupciones, o para efectuar funciones de diálogo automático. La descripción completa de

RS1	RS0	CRA 2	CRB 2	REGISTRO SELECCIONADO
0	0	1	-	REGISTRO PERIFÉRICO A
0	0	0	-	REGISTRO DIRECC. DATOS A
0	1	-	-	REGISTRO CONTROL A
1	0	-	1	REGISTRO PERIFÉRICO B
1	0	-	0	REGISTRO DIRECC. DATOS B
1	1	-	-	REGISTRO CONTROL B

Figura 7-3 Direccionamiento de los registros de un PIO.

los medios proporcionados no es necesaria en este punto. Simplemente, el usuario de cualquier sistema concreto que utilice el 6520 tendrá que referirse a la hoja de características técnicas, que muestra el efecto producido por los diferentes bits del registro de control. Siempre que se inicialice el sistema, el programador tendrá que cargar el registro de control del 6520 con el contenido correcto para la aplicación prevista.

El 6530

El 6530 realiza una combinación de cuatro funciones: RAM, ROM, PIO y TIMER (temporizador). La RAM es una memoria de 64 × 8. La ROM

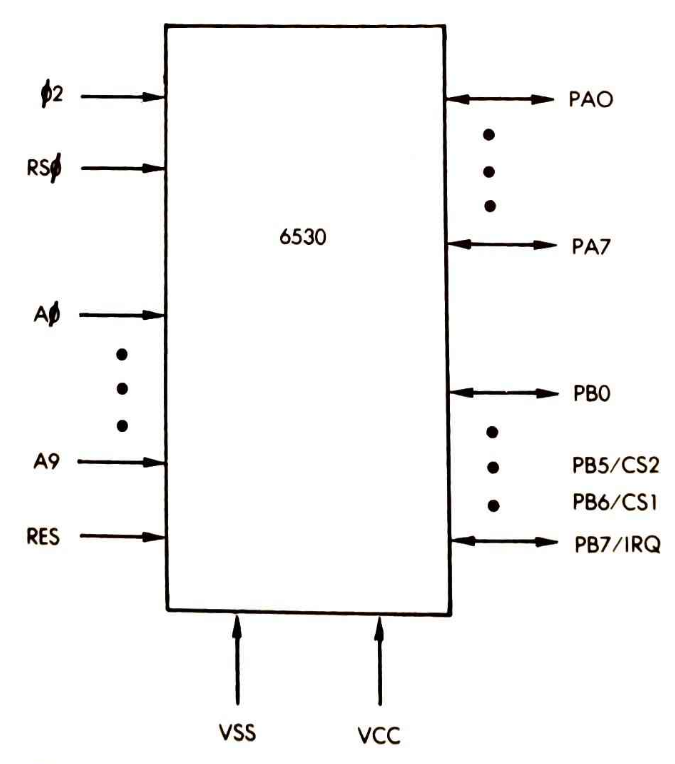


Figura 7-4 Diagrama de conexiones del 6530.

es una memoria de 1 K×8. El temporizador proporciona al programador múltiples posibilidades de realización de retardos internos. La sección de PIO es, en lo esencial, análoga al 6520, que hemos descrito. Hay dos "ports", cada uno con un registro de datos y un registro de dirección de datos. Un "0" en una posición de bit dada del registro de dirección especifica una entrada, mientras que un "1" especifica una salida.

El temporizador de intervalos programable puede programarse para contar hasta 256 intervalos (puesto que tiene internamente 8 bits). El programador puede especificar un período de tiempo de 1, 8, 64 o 1024 veces el período de reloj del sistema. Cuando se haya acabado el contaje, el indicador de interrupción de la pastilla ("chip") se pondrá a un nivel lógico "1". El contenido del temporizador se establece por medio del bus de datos. Los cuatro intervalos de tiempo posibles deben especificarse en las líneas A0 y A1 del bus de dirección.

Tres terminales del "port" B tienen una función doble: PB5, PB6 y PB7 pueden utilizarse para funciones de control. El terminal PB7, por ejemplo, puede programarse como terminal de entrada de petición de interrupción.

Esta pastilla se utiliza, en particular, en la placa KIM (Observación: En KIM, el terminal PB6 no está disponible).

Programación de un PIO

A título de ejemplo, veamos un programa para utilizar en un 6520 o un 6522 (del que suponemos que el registro de control ya está posicionado).

```
LDA   #FF    PREPARAR EL REGISTRO DE DIRECCIÓN
STA   DDRB   CONFIGURAR EL PORT B EN SALIDA
LDA   #00
STA   IORB   GENERAR UNA SALIDA CERO
```

DDRB es la dirección del registro de dirección de los datos del "port" B para este PIO. IORB es registro de datos o E/S del "port" B, "FF" en hexadecimal es "11111111" en binario = todas las salidas.

El 6522

El 6522, llamado también VIA (versatile interface adapter-adaptador de interface versátil) es una versión perfeccionada del 6520. Además de las posibilidades del 6520, tiene dos temporizadores de intervalos programables, un convertidor serie-paralelo y paralelo-serie y la posibilidad de enclavar los datos de entrada.

La descripción detallada del hardware de este componente queda fuera

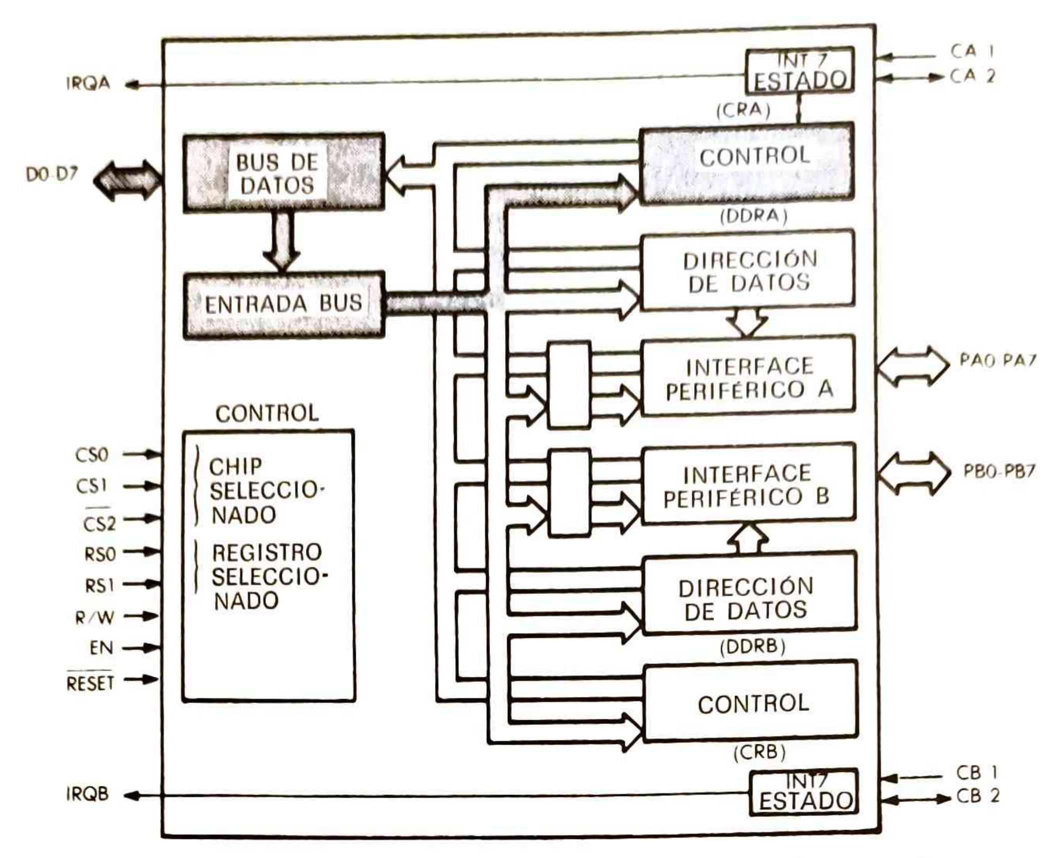


Figura 7-5 Empleo de un PIO: carga del registro de control.

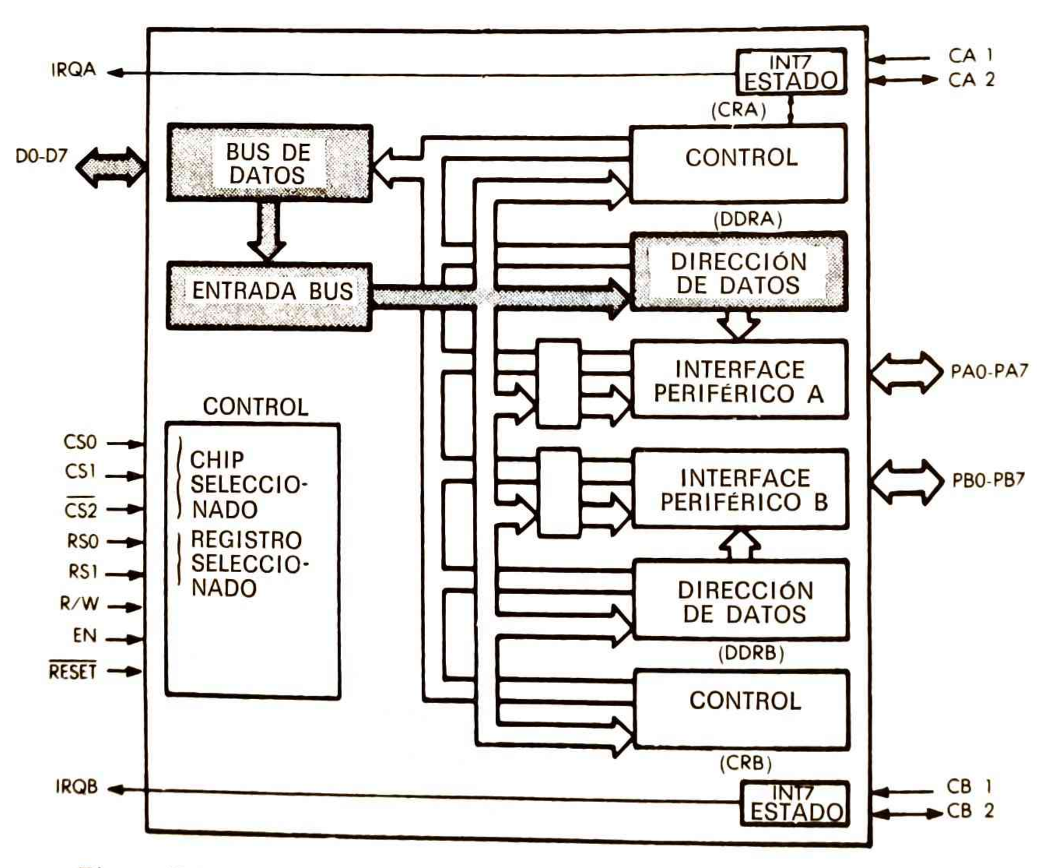


Figura 7-6 Empleo de un PIO: carga de la dirección de datos.

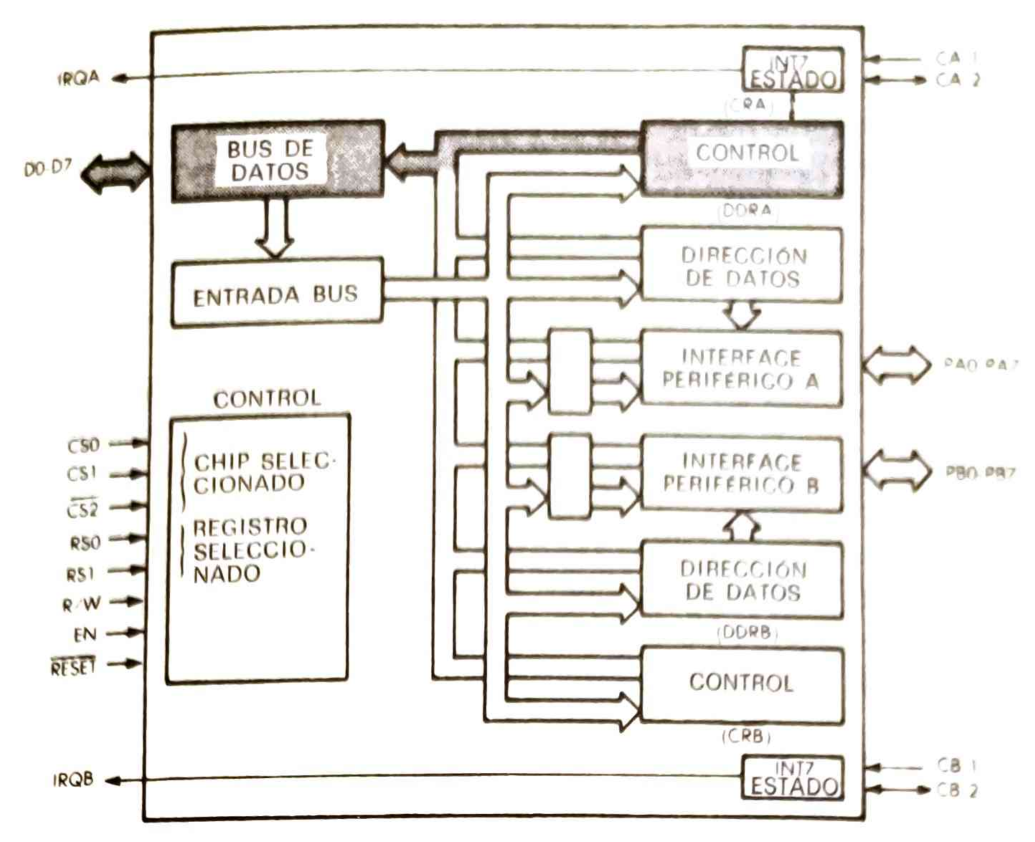


Figura 7-7 *Empleo de un PIO: lectura del estado.*

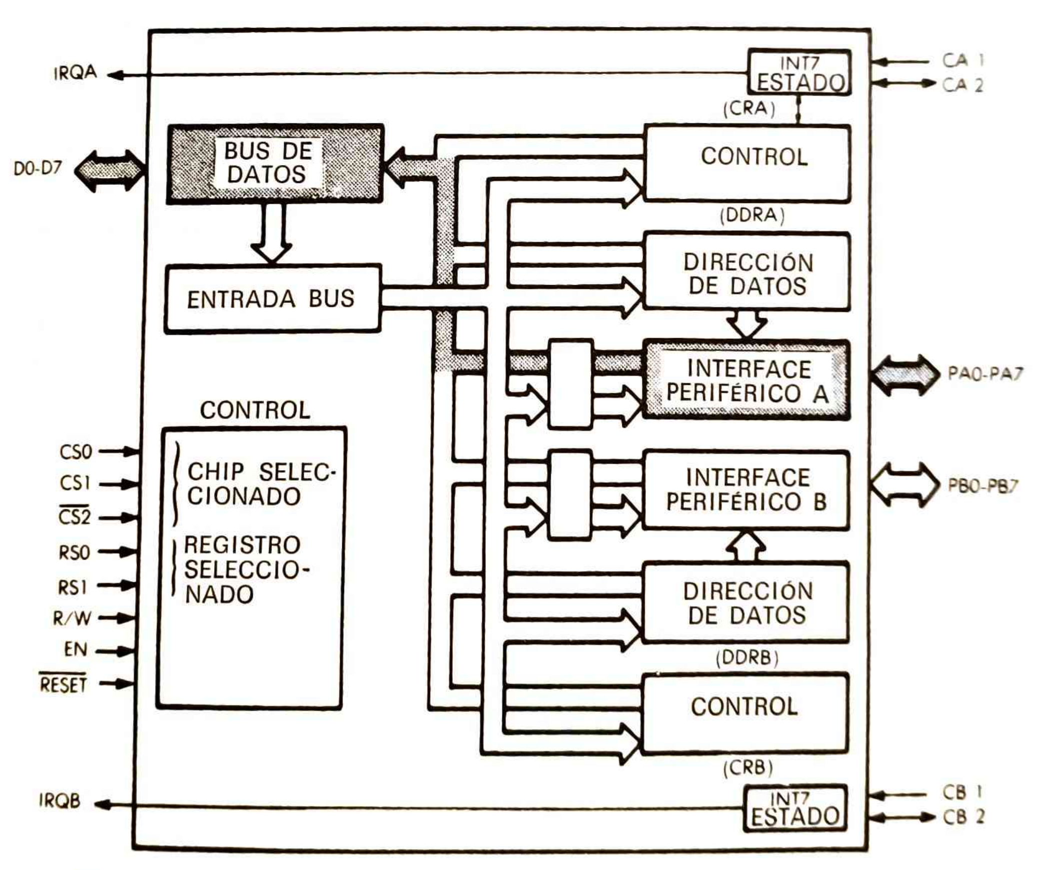


Figura 7-8 *Empleo de un PIO: lectura de datos de entrada.*

del objeto de este libro. Simplemente, con la ayuda de la descripción que se ha dado para los componentes anteriores, debe ser fácil para el programador familiarizarse por sí mismo con el direccionamiento de los registros internos de este componente, así como con su programación. Esta información se proporciona en las hojas de características técnicas del fabricante.

El 6532

El 6532 es una pastilla integrada combinada que comprende una RAM de 128 × 8, un PIO con dos "ports" bidireccionales y un temporizador de intervalos programables. Se utiliza en la tarjeta circuital SYM, fabricada por Synertek Systems, que es análoga a la tarjeta KIM, fabricada por MOS Technology y por Rockwell.

Una vez más, el usuario debe examinar cuidadosamente las hojas de características técnicas de este componente para aprender a direccionar y utilizar los diversos registros internos.

RESUMEN

Lamentablemente, para utilizar eficazmente dichos componentes, será necesario comprender a fondo la función de cada bit, o de cada grupo de bits, en los diferentes registros de control. Estas nuevas pastillas complejas automatizan un cierto número de procedimientos que antes tenían que realizarse por software o lógica especial. En particular, muchos de los procedimientos de diálogo están automatizados en componentes tales como el 6522. Además, se puede encontrar en el interior algo de tratamiento y detección de las interrupciones. Con la información que se ha presentado en el capítulo anterior, el lector debe ser capaz de examinar las hojas de características técnicas correspondientes y de comprender las funciones de las diversas señales y registros. Naturalmente, nuevos componentes están siempre a punto de introducirse y proporcionarán medios para realización por hardware de algoritmos todavía más complejos.

8 Ejemplos de aplicación

INTRODUCCIÓN

Este capítulo tiene por objeto probar las aptitudes del lector para programar presentando una serie de programas de utilidad. Estos programas, o "rutinas", suelen encontrarse en las aplicaciones habituales y por ello, se les denomina "rutinas de utilidad". Requieren una síntesis de los conocimientos y de las técnicas hasta ahora presentadas.

Vamos a buscar caracteres a partir de un dispositivo de E/S y tratarlos de diferentes maneras. Pero, en primer lugar, pondremos a cero una zona de memoria (ello no es forzosamente necesario; cada uno de estos programas se presenta solamente a título de ejemplo de programación).

PUESTA A CERO DE UNA ZONA DE MEMORIA

Deseamos poner a cero el contenido de la memoria, desde la dirección BASE + 1 a la dirección BASE + LONGITUD, en donde la longitud es inferior a 256.

El programa es el siguiente:

```
CERO             LDX   #LONGITUD
                 LDA   #0
PONER A CERO     STA   BASE, X
                 DEX
                 BNE   PONER A CERO
                 RTS
```

Obsérvese que el registro X se utiliza como índice para apuntar a la posición corriente de la zona de memoria a poner a cero.

El acumulador A se carga, de una vez por todas, con el valor "0" (todos los bits a "0") y luego se escribe en las posiciones de memoria sucesivas: BASE + LONGITUD, BASE + LONGITUD − 1, etc., hasta que X se decrementa a cero. Cuando X = 0, se efectúa el retorno del subprograma.

Este programa podría servir, por ejemplo, en una prueba de memoria en donde se pondría un bloque a cero y luego se verificaría su contenido.

Ejercicio 8.1: *Escribir un programa de prueba de memoria que ponga a cero un bloque de 256 bytes (palabras) y verifique que cada posición es 0. A continuación, se escriben todos los bits a "1" y se verifica el contenido del bloque. Luego, se escribe "01010101" y se verifica el contenido. Finalmente, se escribe "10101010" y se verifica.*

Ahora, hagamos un escrutinio interrogando a nuestros periféricos para determinar cuáles necesitan una intervención de servicio.

ESCRUTINIO DE PERIFÉRICOS

Supondremos que hay 3 dispositivos de E/S (periféricos) conectados a nuestro sistema. Sus registros de estado están situados en las direcciones ESTADO1, ESTADO2 y ESTADO3.

Si sus bits de estado están en la posición 7, sólo tendremos que leer los registros de estado y probar sus bits de signo. Si los bits de estado están en una posición distinta a 7, tendremos que utilizar la instrucción BIT del 6502:

```
TEST    LDA   MÁSCARA
        BIT   ESTADO1
        BNE   ENCONTRADO1
        BIT   ESTADO2
        BNE   ENCONTRADO2
        BIT   ESTADO3
        BNE   ENCONTRADO3
        (salida si no se encuentra)
```

La MÁSCARA contendrá, por ejemplo, "00100000" si probamos la posición 5. La instrucción BIT tiene por resultado poner a cero el bit Z si "MÁSCARA Y ESTADO" no es cero, es decir, si el bit correspondiente de ESTADO está de acuerdo con el bit de MÁSCARA. La instrucción BNE (bifurcación si no es igual a cero) dará lugar a una bifurcación a la rutina ENCONTRADO adecuada.

LECTURA DE CARACTERES

Supongamos que acabamos de encontrar que un carácter está dispuesto en el teclado. Acumulemos los caracteres en una zona de memoria denominada buffer hasta que encontremos un carácter llamado SPC, cuyo código se ha definido anteriormente.

El subprograma LEERCAR buscará un carácter a partir del teclado (para más detalles, ver capítulo 6) y lo deja en el acumulador. Suponemos que se busca un máximo de 256 caracteres antes de que se encuentre un carácter SPC.

```
CADENA      LDX   #0          INICIALIZAR ÍNDICE A CERO
SIGUIENTE   JSR   LEERCAR
            CMP   #SPC        ¿ES EL CARÁCTER DE INTERRUP-
                              CIÓN?
            BEQ   FIN         SI ES ASÍ, SE ACABA
            STA   BUFFER, X   SI NO, SALVAR EL CARÁCTER
            INX               INCREMENTAR PUNTERO
            JMP   SIGUIENTE   TOMAR EL PRÓXIMO CARÁCTER
FIN         RTS
```

Ejercicio 8.2: *Mejoremos la rutina de base:*
a) reenviar el carácter en eco (para un teletipo por ejemplo);
b) verificar que la cadena de entrada no es más larga que 256 caracteres.

Ahora tenemos una cadena de caracteres en un buffer de memoria. Efectuemos diferentes tratamientos de dicha cadena.

PRUEBA DE UN CARÁCTER

Determinemos si el carácter situado en la dirección LOC es igual a 0, 1 o 2:

```
C12   LDA   LOC
      CMP   #$00
      BEQ   CERO
      CMP   #$01
      BEQ   UNO
      CMP   #$02
      BEQ   DOS
      JMP   NO ENCONTRADO
```

Leemos simplemente el carácter y luego utilizamos la instrucción CMP para verificar su valor.

Efectuemos ahora una prueba diferente.

PRUEBA EN UN INTERVALO

Determinemos si el carácter ASCII en la posición de memoria LOC es un dígito comprendido entre 0 y 9:

```
INTERV    LDA    #$40
          ADC    #$40       FORZAR DESBORDAMIENTO
          LDA    LOC
          ORA    #$80       PONER BIT 7 = 1
          CMP    #$B0       ASCII0
          BCC    INF
          CMP    #$B9       ASCII9
          BEQ    SALIDA     9 EXACTAMENTE
          BCS    SUP
SALIDA    CLC               PUESTA A 0 DE C
          CLV               PUESTA A 0 DE V
          RTS
INF       SEC               PUESTA A 1 DE C
          CLV
          RTS
SUP       RTS               (C Y V ESTÁN A 1)
```

ASCII0 está representado en hexadecimal por "B0"
ASCII9 está representado en hexadecimal por "B9"

Recuérdese que cuando se utiliza una instrucción CMP, el bit de acarreo se pondrá a "1" si el valor del literal que sigue es inferior o igual al contenido del acumulador. Se pondrá a "0" si dicho literal es superior.

Si B0 es superior al carácter, nuestro carácter está por debajo del intervalo y se producirá una bifurcación. Si no es superior, lo comparamos con B9. Si es menor o igual a 9, todo es correcto y saldremos. De no ser así, pasaremos a SUP.

Cuando salgamos de este programa, queremos saber si el carácter está por debajo o por encima o comprendido en el intervalo de 0 a 9. Ello se indicará por los bits C y V. V no es alterado por CMP, mientras que sí cambia a Z, N y C.

Cuando se retorna desde este subprograma, un "0" en V indica que se

está por encima del intervalo, un "1" en C indica que se está por debajo de dicho intervalo y un "0" en C indica un dígito correcto entre 0 y 9.

Naturalmente, podrían utilizarse otros convenios, tales como cargar un dígito en el acumulador para indicar el resultado de las pruebas.

Ejercicio 8.3: *Simplificar el programa anterior haciéndolo probar con respecto al carácter ASCII que sigue a 9, en lugar de probar con respecto al 9 exactamente.*

Ejercicio 8.4: *Determinar si un carácter ASCII contenido en el acumulador es una letra del alfabeto.*

Cuando se utiliza una tabla ASCII, se observa que suele emplearse la *paridad* (en el ejemplo anterior no se utiliza la paridad). Por ejemplo, el código ASCII de "0" es "0110000" (código de 7 bits). Sin embargo, si utilizamos, por ejemplo, la paridad impar (garantizamos que el número total de "unos" de una palabra es impar), entonces, el código se hace "10110000". Un "1" suplementario se añade a la izquierda. Esto es "B0" en hexadecimal. Desarrollemos, pues, un programa para generar la paridad.

GENERACIÓN DE PARIDAD

Este programa generará una paridad par en la posición de bit 7:

```
PARIDAD     LDX   #$07        CONTAJE DE LOS BITS
            LDA   #$00
            STA   CONT1       CONTAJE DE LOS "1"
            LDA   CAR         LEER EL CARÁCTER
            ROL   A           ELIMINAR EL BIT 7
SIGUIENTE   ROL   A           BIT SIGUIENTE
            BCC   CERO        ¿ES UN "1"?
UNO         INC   CONT1
CERO        DEX               DECREMENTAR EL CONTAJE
                              DE BITS
            BNE   SIGUIENTE   ¿EL ÚLTIMO BIT?
            ROL   A           RESTAURAR EL BIT 0
            ROL   A           ELIMINAR EL BIT 7
            LSR   CONT1       EL BIT MÁS A LA DERECHA
                              ES LA PARIDAD
            ROR   A           PONERLO EN A
            RTS
```

Se utiliza el registro X para contar los bits a medida que se desplazan a la izquierda del acumulador. Cada vez que se desplaza un "1" a la izquierda de A (se prueba por BCC), se incrementa el contador de "unos". Cuando se hayan desplazado 8 bits (el programa ignora el bit 7 que será el bit de paridad), se desplaza A a la izquierda dos veces más para llevar el bit 6 a la izquierda de A.

El bit de paridad correcto es el bit más a la derecha de CONT1; se envía en el indicador de acarreo por medio de LSR y se hace el bit 7 de A. Otro ROR A copia este bit en la posición 7 de A y se ha acabado el programa.

Ejercicio 8.5: *Con el empleo del programa anterior de ejemplo, verificar la paridad de una palabra. Debe calcular la paridad correcta y luego compararla con la prevista.*

CONVERSIÓN DE CÓDIGO: ASCII A BCD

La conversión de ASCII a BCD es muy sencilla. Observaremos que la presentación hexadecimal de los caracteres ASCII 0 a 9 es 30 a 39 sin paridad o bien B0 a B9 con paridad. La representación en BCD se obtiene simplemente eliminando el 3 o la B respectivamente; es decir, enmascarado el "nibble" (cuarteto) de la izquierda:

```
LDA    CAR
AND    #$OF       ENMASCARAMIENTO DEL NIBBLE DE LA IZQ.
STA    BCDCAR
```

Ejercicio 8.6: *Escribir un programa para convertir BCD a ASCII.*

Ejercicio 8.7: *(más difícil). Escribir un programa para convertir BCD a binario.*

Recomendación: N_3 N_2 N_1 N_0 en BCD es $\{[(N_3 \times 10)+N_2] \times 10+N_1\} \times$ $\times$ $10 + N_0$ en binario.

Para multiplicar por 10, utilícese un desplazamiento a la izquierda $(=\times 2)$, otro desplazamiento a la izquierda $(=\times 4)$, un ADC $(=\times 5)$ y otro desplazamiento a la izquierda $(=\times 10)$.

En notación BCD completa, la primera palabra puede contener el número de dígitos BCD, el siguiente "nibble" puede contener el signo y cada "nibble" (4 bits) sucesivo puede contener un dígito BCD (suponemos la ausencia de decimales). El último "nibble" del bloque puede ser inutilizado.

ENCONTRAR EL ELEMENTO MÁS GRANDE DE UNA TABLA

La dirección de comienzo de la tabla está contenida en la dirección de memoria BASE en página cero. El primer elemento de la tabla es el número de bytes que contiene. Este programa busca el elemento más grande de la tabla. El valor se dejará en el acumulador A y su posición se almacenará en la posición de memoria INDEX.

Este programa utiliza los registros A e Y y se utilizará direccionamiento indirecto, de modo que pueda tratar cualquier tabla en cualquier lugar en la memoria.

```
MAX        LDY   #0           PUNTERO HACIA LA TABLA
           LDA   (BASE), Y    ACCESO AL ELEMENTO 0
                              = LONGITUD
           TAY                CONSERVAR LONGITUD EN Y
           LDA   #0           INICIALIZACIÓN A CERO DEL
                              VALOR MÁXIMO
           STA   INDEX        INICIALIZACIÓN A CERO DEL
                              ÍNDICE
BUCLE      CMP   (BASE), Y    ¿ES EL ELEMENTO CORRIENTE
                              EL MÁXIMO?
           BCS   NO CAMBIO    ¿SÍ?
           LDA   (BASE), Y    CARGAR EL NUEVO MÁXIMO
           STY   INDEX        POSICIÓN DEL MÁXIMO
NO CAMBIO  DEY                PUNTERO HACIA EL ELEMEN-
                              TO SIGUIENTE
           BNE   BUCLE        ¿ES PRECISO CONTINUAR?
           RTS                ACABADO SI Y = 0
```

Este programa prueba primero el enésimo elemento. Si es mayor que 0, pasa a A y su posición se memoriza en INDEX. A continuación, se prueba el (N−1)-ésimo elemento, etc.

Este programa funciona para enteros positivos.

Ejercicio 8.8: *Modificar el programa con el fin de que funcione también para números negativos en complemento a dos.*

Ejercicio 8.9: *¿Funcionará también este programa para caracteres ASCII?*

Ejercicio 8.10: *Escribir un programa que clasifique N números en orden creciente.*

Ejercicio 8.11: *Escribir un programa que clasifique N nombres (de 3 caracteres cada uno) en orden alfabético.*

SUMA DE N ELEMENTOS

Este programa calculará la suma de 16 bits de N elementos de una tabla. La dirección de comienzo de la tabla está contenida en la dirección de memoria BASE en página cero. El primer elemento de la tabla contiene el número de elementos N. La suma de 16 bits se dejará en las posiciones de memoria SUMABAJA y SUMAALTA. Si la suma necesitara más de 16 bits, sólo se mantendrán los 16 bits más bajos (se dice que los bits más significativos son objeto de truncación).

El programa modificará los registros A e Y. Supone un máximo de 256 elementos.

```
         LDA  #0         INICIALIZAR SUMA
         STA  SUMABAJA   INICIALIZAR SUMA
         STA  SUMAALTA   INICIALIZAR SUMA
         TAY             INICIALIZAR Y A CERO
         LDA  (BASE), Y  TOMAR N
         TAY             PONERLE EN Y
         CLC             PONER A CERO EL ACARREO
                         PARA ADC
BUCLE    LDA  (BASE), Y  TOMAR EL PRÓXIMO ELEMENTO
         ADC  SUMABAJA   AÑADIRLO A SUMABAJA
         STA  SUMABAJA   CONSERVAR RESULTADO
         BCC  SINACAR    ¿ACARREO?
         INC  SUMAALTA   AÑADIRLO A SUMAALTA PARA
         CLC             LA PRÓXIMA SUMA
SINACAR  DEY             PASAR SIGUIENTE ELEMENTO
         BNE  BUCLE      CONTINUAR SI Y NO ES CERO
         RTS
```

Este programa es simple y debe ser autoexplicatorio.

Ejercicio 8.12: *Modificar este programa para calcular:*

a) una suma de 24 bits,
b) una suma de 32 bits,
c) detectar cualquier desbordamiento.

CÁLCULO DE UNA SUMA DE CONTROL

Una suma de control ("checksum") es un dígito, o conjunto de dígitos, calculado a partir de un bloque de caracteres sucesivos. La suma de control se calcula en el momento en que los datos se almacenan y se coloca al final. Para verificar la integridad de los datos, la suma de control se vuelve a calcular cuando se leen los datos y se compara con el valor almacenado. Cualquier discrepancia indica un error o una avería.

Se utilizan varios algoritmos. En este caso, vamos a efectuar la función OR exclusiva de todos los bytes en una tabla de N elementos y dejaremos el resultado en el acumulador. Como es habitual, la base de la tabla se almacena en la dirección BASE en página cero. El primer elemento de la tabla es su número de elementos N. El programa modifica A e Y. N debe ser inferior a 256.

```
SUMACTRL   LDY  #0           APUNTA HACIA EL PRIMER
                             ELEMENTO
           LDA  (BASE), Y    RECUPERA N
           TAY               LO ALMACENA EN Y
           LDA  #0           INICIALIZA SUMA CONTROL
BUCLECTRL  EOR  (BASE), Y    FUNCIÓN OR EXCLUSIVA DEL
                             SIGUIENTE ELEMENTO
           DEY               APUNTA HACIA EL SIGUIENTE
           BNE  BUCLECTRL    CONTINUAR
           RTS
```

CONTAJE DE CEROS

Este programa contará el número de ceros en nuestra tabla habitual y lo dejará en el registro X. Modifica A, X e Y:

```
CEROS   LDY  #0         APUNTA HACIA PRIMER ELEMENTO
        LDA  (DIR), Y   RECUPERA N
        TAY             LO ALMACENA EN Y
        LDX  #0         INICIALIZA EL NÚMERO DE CEROS
BUCLEZ  LDA  (DIR), Y   CONSIGUE ELEMENTO SIGUIENTE
        BNE  NOZ        ¿ES CERO?
        INX             SÍ, CONTARLO
NOZ     DEY             APUNTA HACIA EL SIGUIENTE
        BNE  BUCLEZ     CONTINUAR
        RTS
```

Ejercicio 8.13: *Modificar este programa para contar:*
a) el número de asteriscos (carácter ""),*
b) el número de letras del alfabeto,
c) el número de dígitos entre 0 y 9.

BÚSQUEDA EN UNA CADENA DE CARACTERES

Una cadena de caracteres se almacena en la memoria, según se indica en la figura 8-1. Vamos a recorrer la cadena hasta la aparición de una más corta, o subcadena, llamada plantilla (TEMPLT) de longitud TPTLEN. La longitud de la cadena original es STRLEN y el programa volverá con el registro X, que contiene la posición en donde se encontró TEMPLT o FF, en hexadecimal, si no se encuentra en dicho emplazamiento. En la figura 8-2 se ilustra el diagrama de flujo para este programa. La cadena se recorre primero hasta que se encuentre el primer carácter de TEMPLT. Si nunca se encuentra este primer carácter, el programa se terminará con un fracaso.

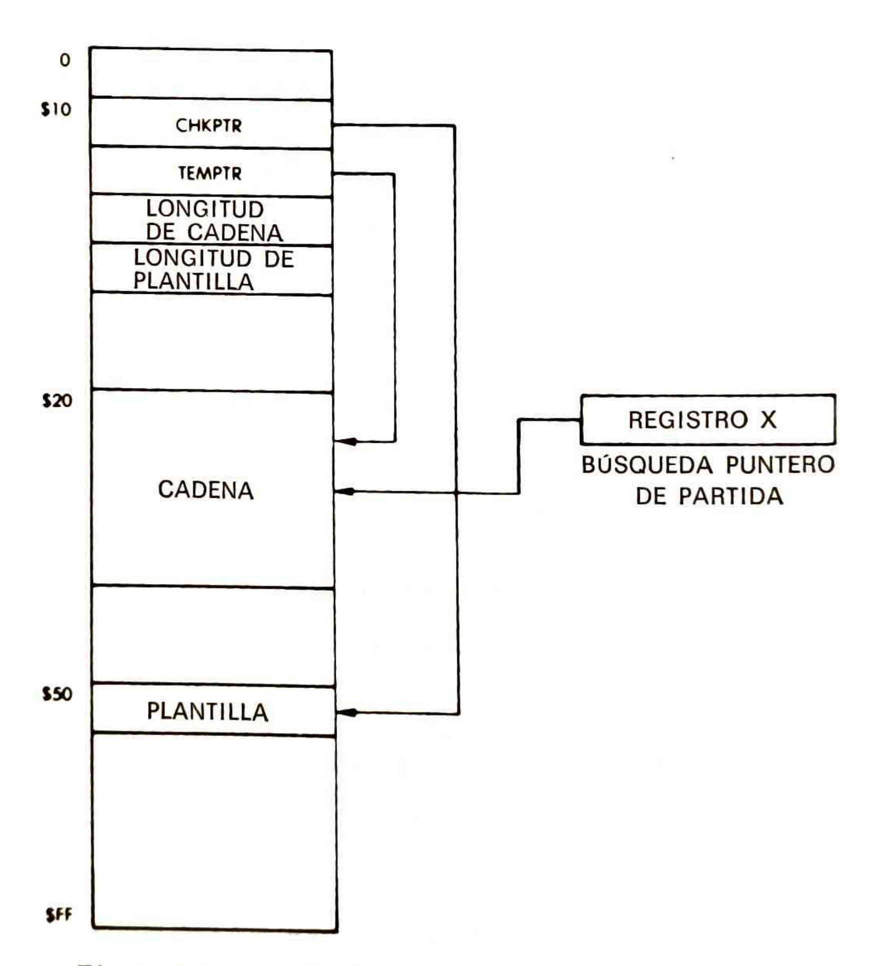


Figura 8-1 Búsqueda en una cadena: la memoria.

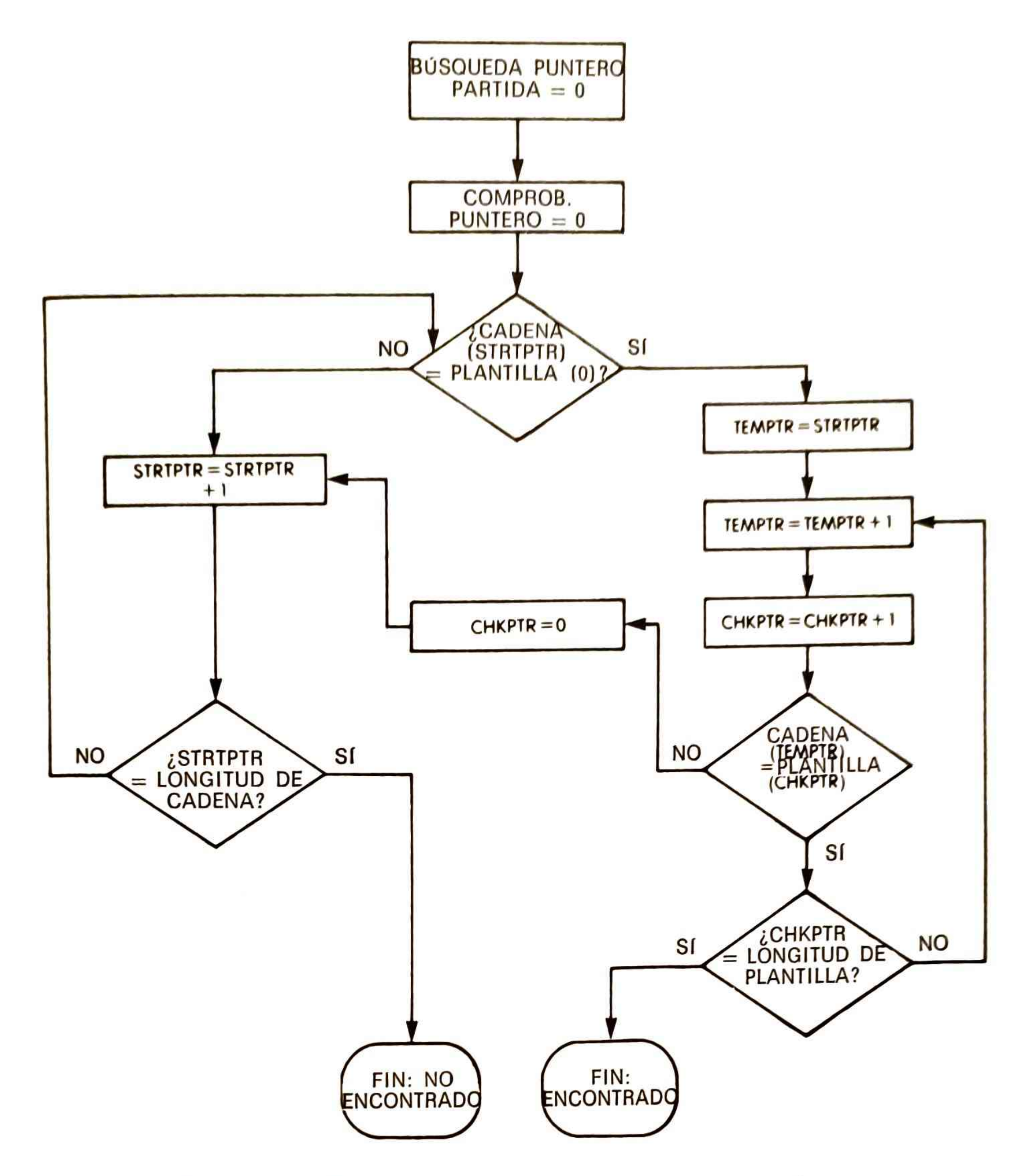


Figura 8-2 Diagrama de flujo: búsqueda en una cadena.

Si se encuentra dicho primer carácter, se compara el segundo carácter con el siguiente en la cadena. Si no hay coincidencia, se reanuda la búsqueda del primer carácter, puesto que podría producirse otra ocurrencia de este primer carácter dentro de la cadena original. Si hay coincidencia en los caracteres primero y segundo, se comparan los siguientes de la misma manera. En la figura **8-3** se ilustra el programa correspondiente. Obsérvese que el registro X sirve de puntero hacia el elemento corriente de la cadena. Naturalmente, se utiliza el direccionamiento indexado para recuperar el elemento de cadena corriente.

```
LOC    CODE        LINE

0000                          ;STRING SEARCH.
0000                          ;FINDS LOCATION IN STRING OF LENGTH 'STRLEN'
0000                          ;STARTING AT 'STRING' OF A TEMPLATE OF
0000                          ;LENGTH 'TPTLEN' STARTING AT 'TEMPLT', AND
0000                          ;RETURNS WITH X=LOCATION OF TEMPLATE
0000                          ;IN STRING IF FOUND, OR X=$FF IF NOT FOUND.
0000                          ;
0000                          STRING = $20           ;1ST LOCATION OF STRING.
0000                          TEMPLT = $50           ;1ST LOCATION OF TEMPLATE.
0000                                 * = $10
0010                          CHKPTR *=*+1
0011                          TEMPTR *=*+1
0012                          STRLEN *=*+1           ;LENGTH OF STRING.
0013                          TPTLEN *=*+1           ;LENGTH OF TEMPLATE.
0014                                 * = $200
0200   A2 00                         LDX #0          ;RESET SEARCH START POINTER.
0202   A5 50                  NXTPOS LDA TEMPLT      ;IS FIRST ELEMENT OF TEMPLATE...
0204   D5 20                         CMP STRING,X    ;= CURRENT STRING ELEMENT?
0206   F0 08                         BEQ CHECK       ;IF YES, CHECK FOR REST OF MATCH.
0208   E8                     NXTSTR INX             ;INCREMENT SEARCH START COUNTER.
0209   E4 12                         CPX STRLEN      ;IS IT EQUAL TO STRING LENGTH?
020B   D0 F5                         BNE NXTPOS      ;NO, CHECK NEXT STRING POSITION.
020D   A2 FF                         LDX #$FF        ;YES, SET 'NOT FOUND' INDICATOR.
020F   60                            RTS             ;RETURN: ALL CHRS CHECKED.
0210   86 11                  CHECK  STX TEMPTR      ;LET TEMPORARY POINTER=
0212                                 ;CURRENT STRING POINTER.
0212   A9 00                         LDA #0
0214   85 10                         STA CHKPTR      ;RESET TEMPLATE POINTER.
0216   E6 11                  CHKLP  INC TEMPTR      ;INCREMENT TEMPORARY POINTER.
0218   E6 10                         INC CHKPTR      ;INCREMENT TEMPLATE POINTER.
021A   A4 10                         LDY CHKPTR
021C   C4 13                         CPY TPTLEN      ;DOES TEMPLATE POINTER=TEMPLATE LENGTH?
021E   F0 0C                         BEQ FOUND       ;IF YES, TEMPLATE MATCHED.
0220   B9 50 00                      LDA TEMPLT,Y    ;LOAD TEMPLATE ELEMENT.
0223   A4 11                         LDY TEMPTR
0225   D9 20 00                      CMP STRING,Y    ;COMPARE TO STRING CHR.
0228   D0 DE                         BNE NXTSTR      ;IF NO MATCH, CHECK NEXT STRING CHR.
022A   F0 EA                         BEQ CHKLP       ;IF MATCH, CHECK NEXT CHR.
022C   60                     FOUND  RTS             ;DONE.
022D                                 .END
```

Figura 8-3 Programa de búsqueda en una cadena.

RECAPITULACIÓN

En este capítulo, hemos presentado rutinas de utilidad ordinarias que utilizan combinaciones de las técnicas descritas en capítulos anteriores. Estas rutinas deben permitir comenzar a escribir los propios programas. Muchos de ellos han utilizado una estructura de datos especial, la tabla. Sin embargo, existen otras posibilidades para estructurar los datos y serán objeto de revisión en el siguiente capítulo.

9 Estructuras de datos

1.ª PARTE. CONCEPTOS DE DISEÑO

INTRODUCCIÓN

La concepción de un buen programa lleva consigo dos tareas: *el desarrollo del algoritmo y la elección de las estructuras de datos.* En la mayoría de los programas sencillos, ninguna estructura de datos significativa interviene, de modo que el problema principal que se debe superar para aprender la programación es aprender a diseñar algoritmos y codificarlos eficazmente en un lenguaje máquina dado. Esto es lo que hemos realizado hasta aquí. No obstante, la concepción de programas más complejos requiere también una comprensión de estructuras de datos. Dos estructuras de datos se han utilizado ya a lo largo del libro: la tabla y la pila. El propósito de este capítulo es presentar otra estructura de datos más general que se desee utilizar. Este capítulo es completamente independiente del microprocesador e incluso del ordenador seleccionado. Es teórico y se refiere a la organización lógica de datos en el sistema. Existen libros especializados sobre estructuras de datos, del mismo modo que existen libros especializados sobre el tema de multiplicaciones eficaces, divisiones u otros algoritmos habituales. En consecuencia, se debe considerar este capítulo como un resumen y se limitará necesariamente a lo esencial. No pretende ser exhaustivo. Revisemos ahora las estructuras de datos más comunes.

PUNTEROS

Un puntero es un número que se utiliza para designar la posición del

dato real. Todo puntero es una dirección. Sin embargo, toda dirección no se llama necesariamente puntero. Una dirección es un puntero solamente si apunta a un cierto tipo de datos o hacia una información estructurada. Hemos encontrado ya un puntero típico, el puntero de pila, que apunta a la cima de la pila (o generalmente, justamente por encima de la pila). Veremos que la pila es una estructura de datos común, que se llama estructura LIFO.

Como ejemplo, cuando se utiliza el direccionamiento indirecto, la dirección indirecta es siempre un puntero hacia los datos que se desea recuperar.

Ejercicio 9.1: *Examine la figura 9-1. En la dirección 15 de la memoria hay un puntero hacia la tabla T. La tabla T comienza en la dirección 500. ¿Cuál es el contenido exacto del puntero hacia T?*

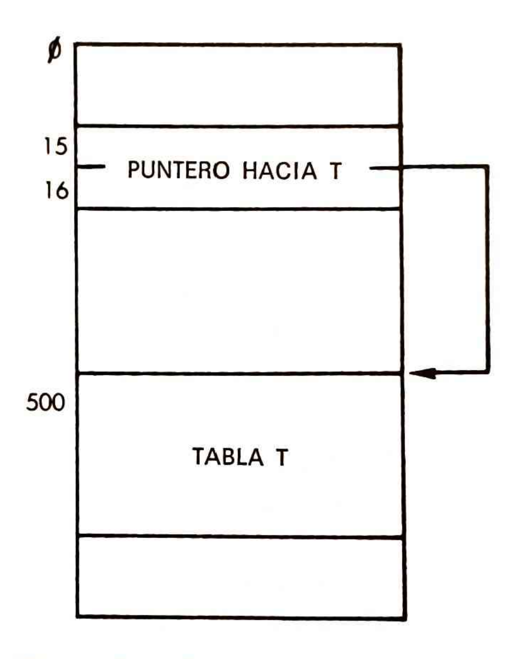


Figura 9-1 Puntero de indirección.

LISTAS

Casi todas las estructuras de datos están organizadas como listas de diversas clases.

Listas secuenciales

Una lista secuencial, tabla o bloque, es probablemente la estructura de datos más simple y que ya hemos utilizado. Las tablas suelen estar ordena-



das en función de un criterio específico, tal como, por ejemplo, orden alfabético u orden numérico.

Es fácil, entonces, recuperar un elemento en un tabla, utilizando, por ejemplo, direccionamiento indexado, tal como hemos hecho. Un bloque suele designar a un grupo de datos que tiene límites definidos pero cuyos contenidos no están ordenados. Puede, por ejemplo, contener una cadena de caracteres, o puede ser un sector o un disco, o puede ser una cierta zona lógica (llamada segmento) de la memoria. En tales casos, no se puede acceder fácilmente a un elemento aleatorio del bloque.

Para facilitar la recuperación de bloques de información, se utilizan directorios.

Directorios

Un directorio es una lista de tablas, o bloques. Por ejemplo, el sistema de ficheros suele utilizar una estructura de directorio. A título de ejemplo sencillo, el directorio maestro del sistema puede incluir una lista de los nombres de usuarios. Esto se ilustra en la figura 9-2. La entrada del usuario "John" apunta al directorio de ficheros de John. Éste es una tabla que contiene los nombres de todos los ficheros de John y su posición. Esto es nuevamente una tabla de punteros. En este caso hemos diseñado un directorio de dos niveles. Un sistema de directorio flexible permitirá la inclusión de

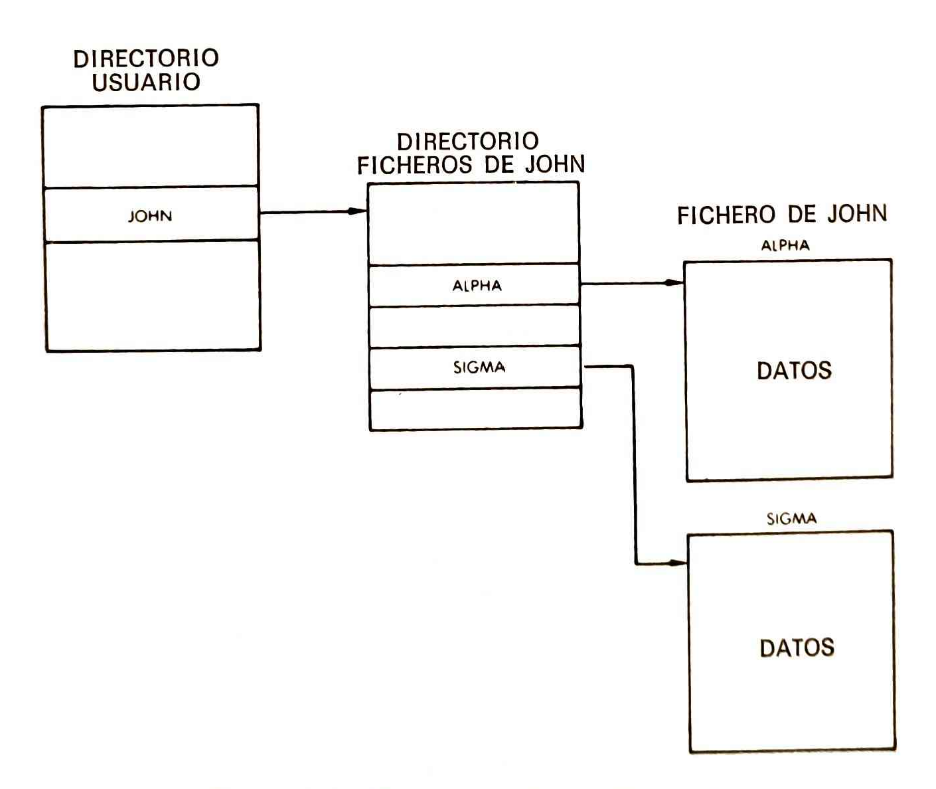


Figura 9-2 Estructura de un directorio.

directorios intermedios adicionales, de modo que puedan ser considerados convenientes por el usuario.

Lista enlazada

En un sistema existen, con frecuencia, bloques de información que representan datos, o sucesos, u otras estructuras, las cuales no se pueden desplazar fácilmente. Si se pudieran desplazar fácilmente, probablemente se agruparían en una tabla para sacarlas o estructurarlas. El problema consiste en que deseamos dejarlas allí donde estén y establecer incluso un orden entre ellas, tales como primera, segunda, tercera o cuarta. Una lista enlazada se utilizará para solucionar este problema. El concepto de lista enlazada se ilustra en la figura 9-3. En la ilustración se observa que un puntero de la lista, llamado PRIMER BLOQUE, apunta al principio del primer bloque. Una posición reservada en el bloque 1 que puede ser, por ejemplo, la primera o última palabra del mismo, contiene un puntero hacia el bloque 2, llamado PTR1. El proceso se repite a continuación para el bloque 2 y el bloque 3. Ya que el bloque 3 es la última entrada de la lista, PTR3, por convenio, contiene un valor especial "nulo" o bien apunta a sí mismo, de modo que se puede detectar el final de la lista. Esta estructura es económica ya que solamente requiere unos pocos punteros (uno por bloque) y permite al usuario evitar la necesidad de desplazar físicamente los bloques en la memoria.



Figura 9-3 Una lista enlazada.

Examinemos, por ejemplo, cómo se inserta un nuevo bloque. Esto se ilustra en la figura 9-4. Supongamos que el bloque nuevo está en la dirección BLOQUE NUEVO y ha de insertarse entre el bloque 1 y el bloque 2. Se cambia simplemente el puntero PTR1 por el valor BLOQUE NUEVO, de modo que ahora apunte al bloque X. PTRX contendrá el valor antiguo de PTR1 (es decir, apuntará al bloque 2). Los otros punteros de la estructura permanecen inalterables. Podemos constatar que la inserción de uno nuevo ha requerido simplemente actualizar dos punteros en la estructura. Esto es eficaz evidentemente.

Ejercicio 9.2: *Dibuje un diagrama que muestre cómo se desplazará el bloque 2 desde esta estructura*

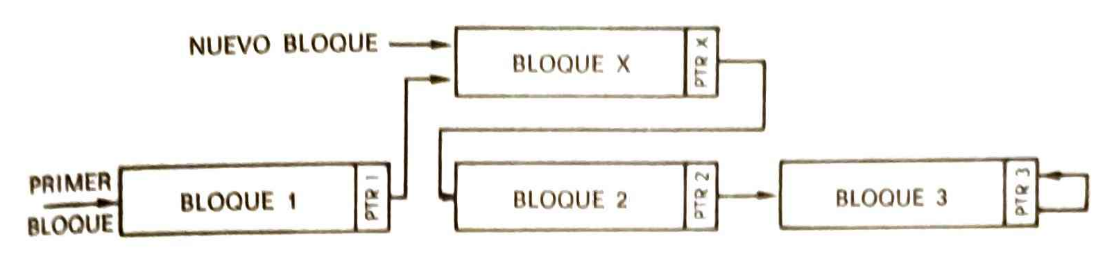


Figura 9-4 Inserción de un bloque nuevo.

Se han desarrollado varias clases de listas para facilitar tipos específicos de acceso, inserciones o supresiones a, o desde, la lista. Veamos algunos de los tipos más frecuentemente empleados de listas enlazadas.

Fila de espera (cola)

La fila de espera de trabajo ("queue") suele llamarse lista FIFO (first-in, first-out, o lista de primero en entrar, primero en salir). Una fila de espera se representa en la figura 9-5. Para aclarar el diagrama podemos suponer, por ejemplo, que el bloque a la izquierda es una rutina de servicio de un dispositivo de salida, tal como una impresora. Los bloques que aparecen a la derecha son los bloques de petición de diversos programas o rutinas, para imprimir caracteres. El orden en el que serán atendidos es el orden establecido por la fila de espera. Se puede ver que el siguiente evento que obtendrá

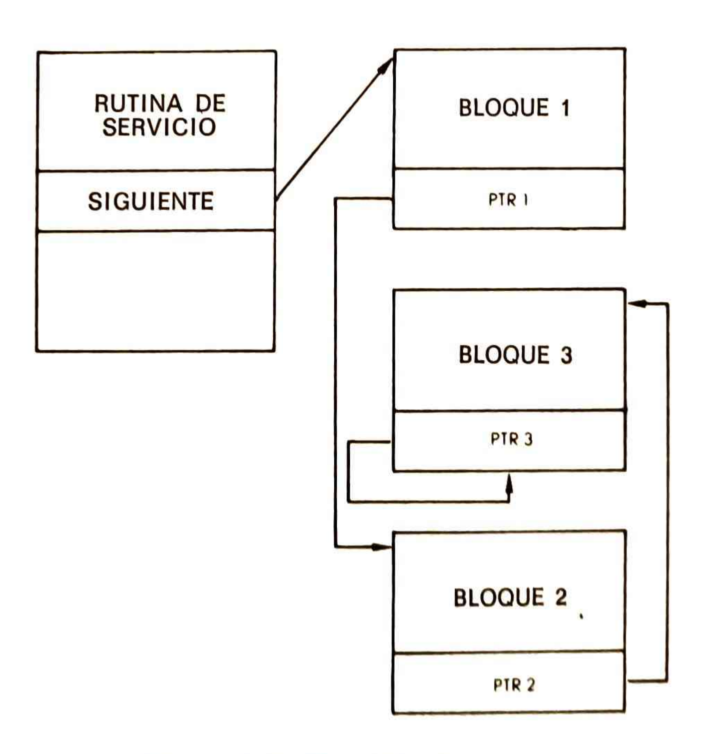


Figura 9-5 Una fila de espera.

servicio es el bloque 1, después el bloque 2 y finalmente, el bloque 3. En una fila de espera, el convenio es que cualquier suceso nuevo que llegue a la misma, se insertará al final de ella. En este caso se insertará después de PTR3. Esto garantiza que el primer bloque que se haya insertado en la fila de espera será el primero servido. En un sistema de ordenador es bastante frecuente tener filas de espera para un cierto número de sucesos sin esperar a un recurso escaso, tal como el procesador o un dispositivo de entrada/salida.

Pila

La estructura de la pila ya se estudió con detalle a lo largo del libro. Esta es una estructura de último en entrar, primero en salir (LIFO). El último elemento depositado en su cima es el primero que se desplaza. Una pila se puede realizar como un bloque ordenado o bien como una lista. Como la mayoría de las pilas en los microordenadores se utilizan para sucesos de alta velocidad, tales como subprogramas e interrupciones, se asigna un bloque continuo a la pila en vez de utilizar una lista enlazada.

¿Lista enlazada o bloque?

De modo similar, la fila de espera se podrá realizar como un bloque de posiciones reservadas. La ventaja de utilizar un bloque continuo es el acceso rápido y la eliminación de los punteros. La desventaja es que suele ser necesario dedicar un bloque bastante grande para alojar el tamaño de la estructura en el caso más desfavorable. También se hace difícil, o poco práctico, insertar o quitar elementos desde el interior del bloque. Ya que la memoria ha sido siempre un recurso escaso, los bloques se han reservado de modo tradicional para estructuras de tamaño fijo o bien para las estructuras que requieran la máxima velocidad de acceso tal como la pila.

Lista circular

La técnica de secuencias "round robin" es el nombre ordinario de una lista circular, que es una lista enlazada en donde los últimos puntos de

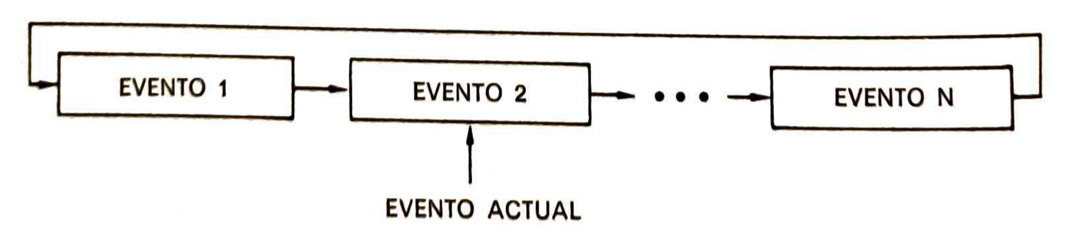


Figura 9-6 "Round-robin" es lista circular.

entrada apuntan de nuevo al primero. Esto se representa en la figura 9-6. En el caso de una lista circular, se suele conservar un puntero de bloque corriente. En el caso de los eventos o programas que esperan servicio, el puntero del suceso corriente se desplazará cada vez una posición a la derecha o a la izquierda. Una lista circular suele corresponder a una estructura en donde se supone que todos los bloques tienen la misma prioridad. Sin embargo, cuando se realiza una exploración, se puede utilizar también una lista circular como un caso particular de otras estructuras simplemente para facilitar el acceso al primer bloque después del último.

Como ejemplo de lista circular, se suele utilizar un programa de escrutinio para realizar un modo secuencial, interrogando a todos los periféricos y luego volviendo de nuevo al primero.

Árboles

Siempre que exista una relación entre todos los elementos de una estructura (que suele llamarse sintaxis), se puede utilizar una estructura de árbol. Un ejemplo sencillo de estructura arbólea, es un árbol de descendientes o árbol genealógico. Esto se representa en la figura 9-7. Se puede ver que Smith tiene dos niños: un hijo Robert y una hija, Jane. Jane a su vez tiene tres niños: Liz, Tom y Phil. Tom, a su vez tiene dos niños: Max y Chris. Sin embargo, Robert a la izquierda de la figura no tiene descendientes.

Esta es una estructura en árbol. De hecho, hemos encontrado ya un ejemplo de árbol sencillo en la figura 9-2. La estructura del directorio es un árbol de dos niveles. El uso de árboles presenta ventajas siempre que los elementos se puedan clasificar de acuerdo con una estructura fija. Ello facilita la inserción y la extracción. Además, los árboles pueden establecer gru-

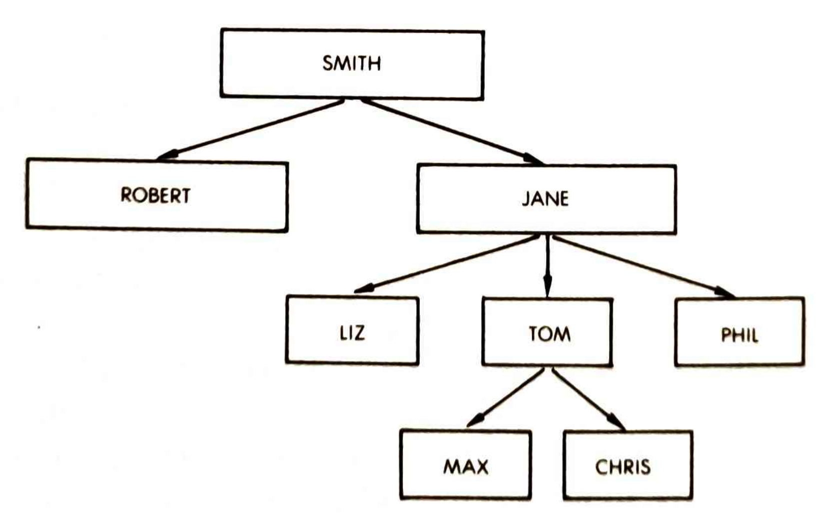


Figura 9-7 Árbol genealógico.

pos de información en un medio estructurado. Tal información puede ser necesaria para procesos posteriores, tal como en un compilador o en un intérprete.

Listas doblemente enlazadas

Se pueden establecer enlaces adicionales entre los elementos de una lista. El ejemplo más simple es la lista doblemente enlazada, ilustrada en la figura 9-8. Se puede ver que tiene la secuencia habitual de enlaces de izquierda a derecha, más otra secuencia de enlaces de derecha a izquierda. El objetivo es permitir la extracción fácil del elemento antes de que haya sido objeto de proceso, así como después del mismo. Eso exige un puntero adicional por bloque.

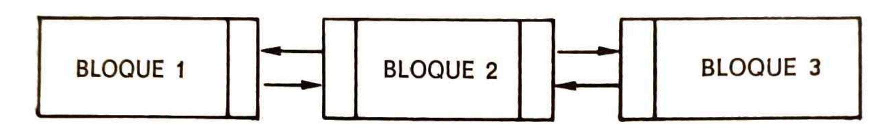


Figura 9-8 Lista doblemente enlazada.

BÚSQUEDA Y CLASIFICACIÓN

La búsqueda y clasificación de los elementos de una lista depende directamente del tipo de estructura que se ha utilizado por la lista. Muchos algoritmos de búsqueda se han desarrollado para las estructuras de datos más frecuentemente utilizadas. Ya hemos utilizado direccionamiento indexado. Esto es posible siempre que los elementos de una tabla estén clasificados de acuerdo a un criterio conocido. Tales elementos pueden ser recuperados por sus números.

La búsqueda secuencial se refiere a la exploración lineal de un bloque completo. Esto es claramente ineficaz pero, en ausencia de una técnica mejor, puede utilizarse cuando los elementos no están clasificados.

La búsqueda binaria o logarítmica trata de encontrar un elemento en una lista clasificada, dividiendo por dos el intervalo de búsqueda en cada paso. Suponiendo, por ejemplo, que estamos buscando en una lista alfabética, se puede comenzar en la mitad de una tabla y determinar si el nombre que buscamos está antes o después de este punto. Si está después de este punto, eliminaremos la primera mitad de la tabla y buscaremos el elemento situado en medio de la segunda mitad. Comparamos de nuevo esta entrada con lo que estamos buscando y limitamos nuestra búsqueda a una de las dos

mitades, y así sucesivamente. La longitud máxima de una búsqueda se garantiza en tal caso que es $\log_2 n$, en donde n es el número de elementos de la tabla.

Existen otras muchas técnicas de búsqueda.

RESUMEN

Esta sección fue concebida solamente como una presentación breve de estructuras de datos típicas que se pueden utilizar por un programador. Aunque la mayoría de las estructuras de datos comunes se han clasificado en clases y se les ha dado un nombre, la organización completa de datos en un sistema complejo puede utilizar cualquier combinación de ellas, u obligar al programador a inventar estructuras más adecuadas. El conjunto de posibilidades solamente está limitado por la imaginación del programador. De modo similar, un número de técnicas de clasificación y búsqueda bien conocidas se han desarrollado para tratar las estructuras de datos habituales. Una descripción completa se sale fuera del marco de este libro. El contenido de esta sección fue concebido para acentuar la importancia del diseño de estructuras de datos adecuadas para los datos que se manejan y proporcionar las herramientas básicas para este fin.

2.ª PARTE. EJEMPLOS DE DISEÑO

INTRODUCCIÓN

Ahora, se presentarán ejemplos de diseños verdaderos para estructuras de datos típicas: tabla, lista enlazada, árbol clasificado. La clasificación, búsqueda y algoritmos de inserción se programarán para estas estructuras. Se describirán también las técnicas avanzadas adicionales tales como clasificación aleatoria y fusión.

Al lector interesado por estas técnicas de programación avanzadas se le recomienda analizar en detalle los programas presentados en esta sección. Sin embargo, el programador principiante puede saltar esta sección inicialmente y volver a la misma cuando se encuentre preparado.

Para sacar provecho de los ejemplos de diseño es necesaria una buena comprensión de los conceptos presentados en la primera parte de este capítulo. Además, en los programas se utilizarán todos los modos de direcciona-

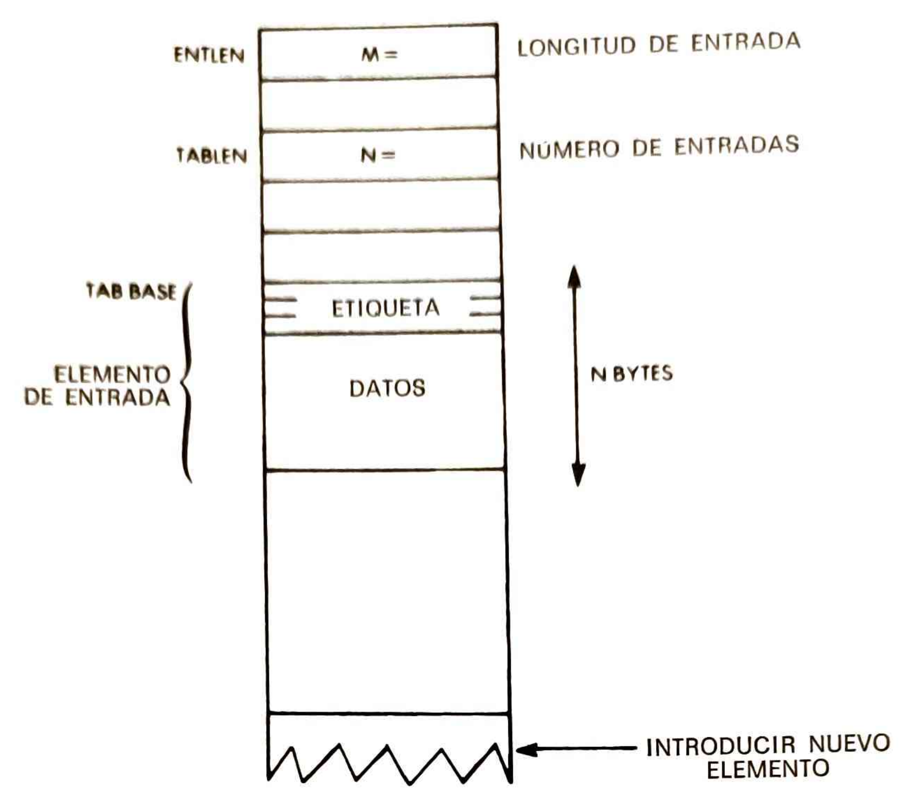


Figura 9-9 La estructura de tablas.

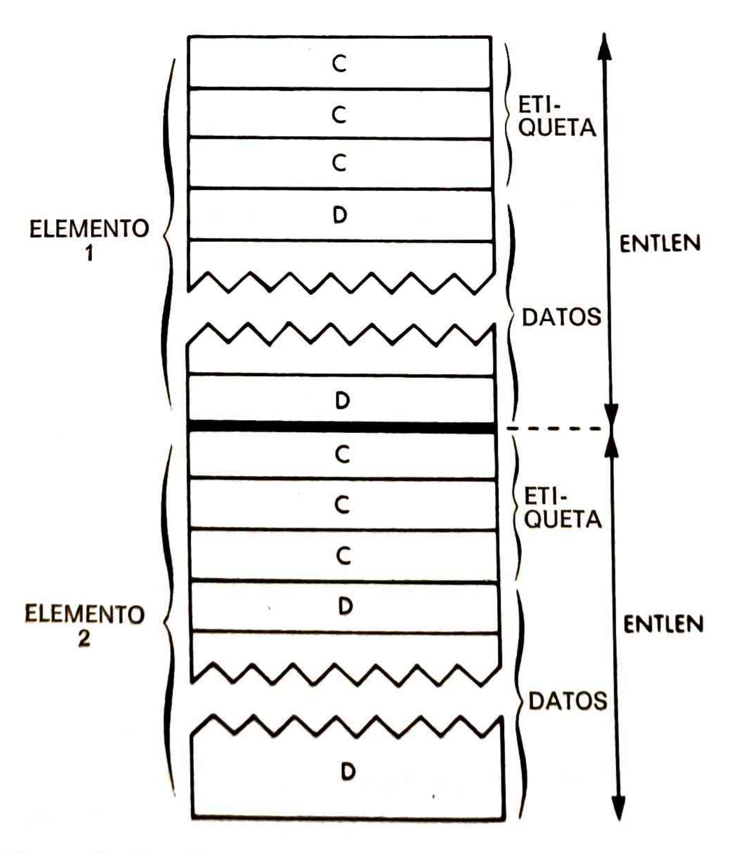


Figura 9-10 Entradas de listas típicas en la memoria.

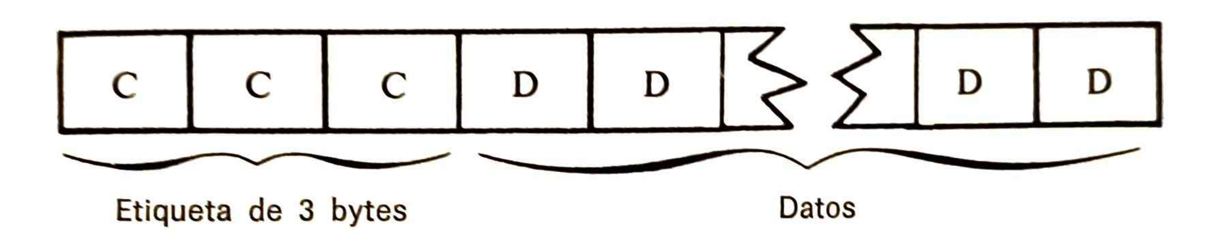
miento del 6502 y se integrarán la mayor parte de los conceptos y técnicas presentadas en los capítulos anteriores.

Vamos a introducir cuatro estructuras: una lista sencilla alfabética, una lista enlazada más directorio y un árbol. Para cada estructura se desarrollarán tres programas: búsqueda, introducción y supresión.

Además se escribirán por separado, al final de la sección, tres algoritmos especializados: clasificación aleatoria, clasificación de burbuja y fusión.

REPRESENTACIÓN DE DATOS EN LA LISTA

La lista sencilla y la lista alfabética utilizarán una representación común para cada elemento de la lista:



Cada elemento o "entrada" incluye una etiqueta de 3 bytes y un bloque de datos de n bytes con n entre 1 y 253. Por tanto, cada entrada utiliza, como máximo, una página (256 bytes). En el interior de cada lista, todos los elementos tienen la misma longitud (figura 9-10). Los programas que trabajan en estas dos listas sencillas utilizan algunas notaciones de variables comunes:

ENTLEN es la longitud de un elemento. Por ejemplo, si cada elemento tiene 10 bytes de datos, ENTLEN = 3 + 10 = 13 bytes.
TABASE es la base de la lista o tabla en memoria.
POINTR es un puntero móvil hacia el elemento en curso.
OBJECT es la entrada objeto, que se va a añadir o suprimir.
TABLEN es el número de entradas (o elementos).

Se supone que son distintas todas las etiquetas. El cambio de esta notación sólo exigirá un cambio pequeño en los programas.

UNA LISTA SENCILLA

La lista sencilla se organiza como una tabla de n elementos. Los elementos no están clasificados (fig. 9-11).

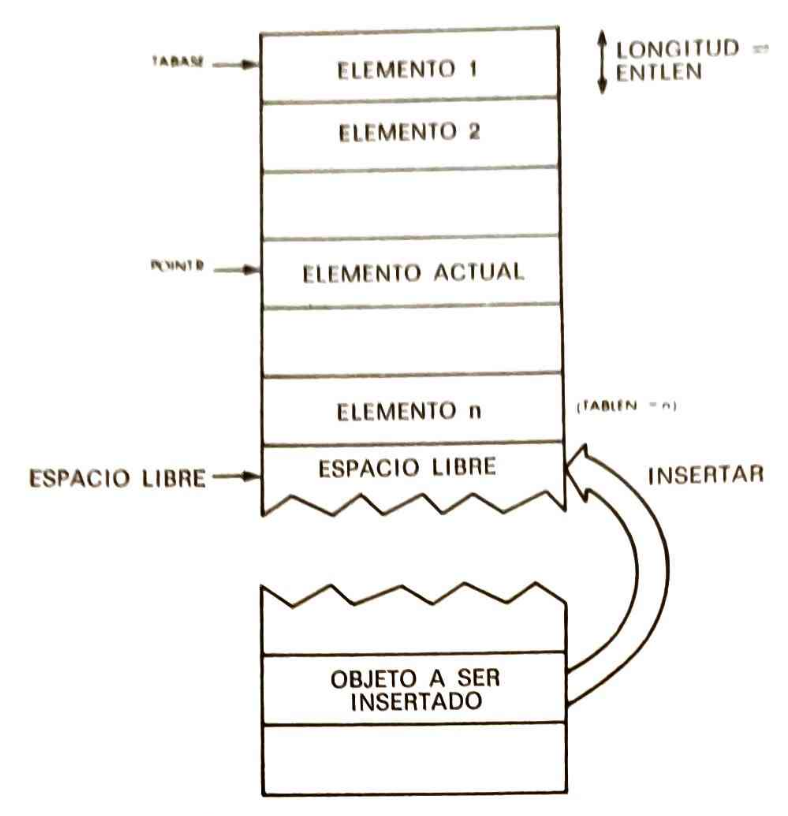


Figura 9-11 La lista simple.

Cuando se realiza una búsqueda, se debe explorar la lista hasta que se encuentra una entrada o se alcance el final de la tabla. Cuando se insertan las nuevas entradas, se añaden a las ya existentes. Cuando se suprime una entrada, se desplazarán hacia abajo, si las hay, las posiciones de memoria más altas, para conservar la continuidad de la tabla.

Búsqueda

Se utiliza una técnica de búsqueda lineal en serie. Cada campo de etiqueta de entrada se compara, letra por letra, con la etiqueta de OBJECT.

El puntero móvil POINTR se inicializa al valor de TABASE.

El registro índice X se inicializa al número de entradas contenidas en la lista (almacenada en TABLEN).

La búsqueda prosigue de modo evidente y el correspondiente diagrama de flujo se muestra en la figura 9-12. El programa se indica en la figura 9-16 al final de esta sección (programa SEARCH).

Inserción de un elemento

Cuando se inserta un nuevo elemento, se utiliza el primer bloque de memoria disponible de (ENTLEN) bytes al final de la lista (fig. 9-11).

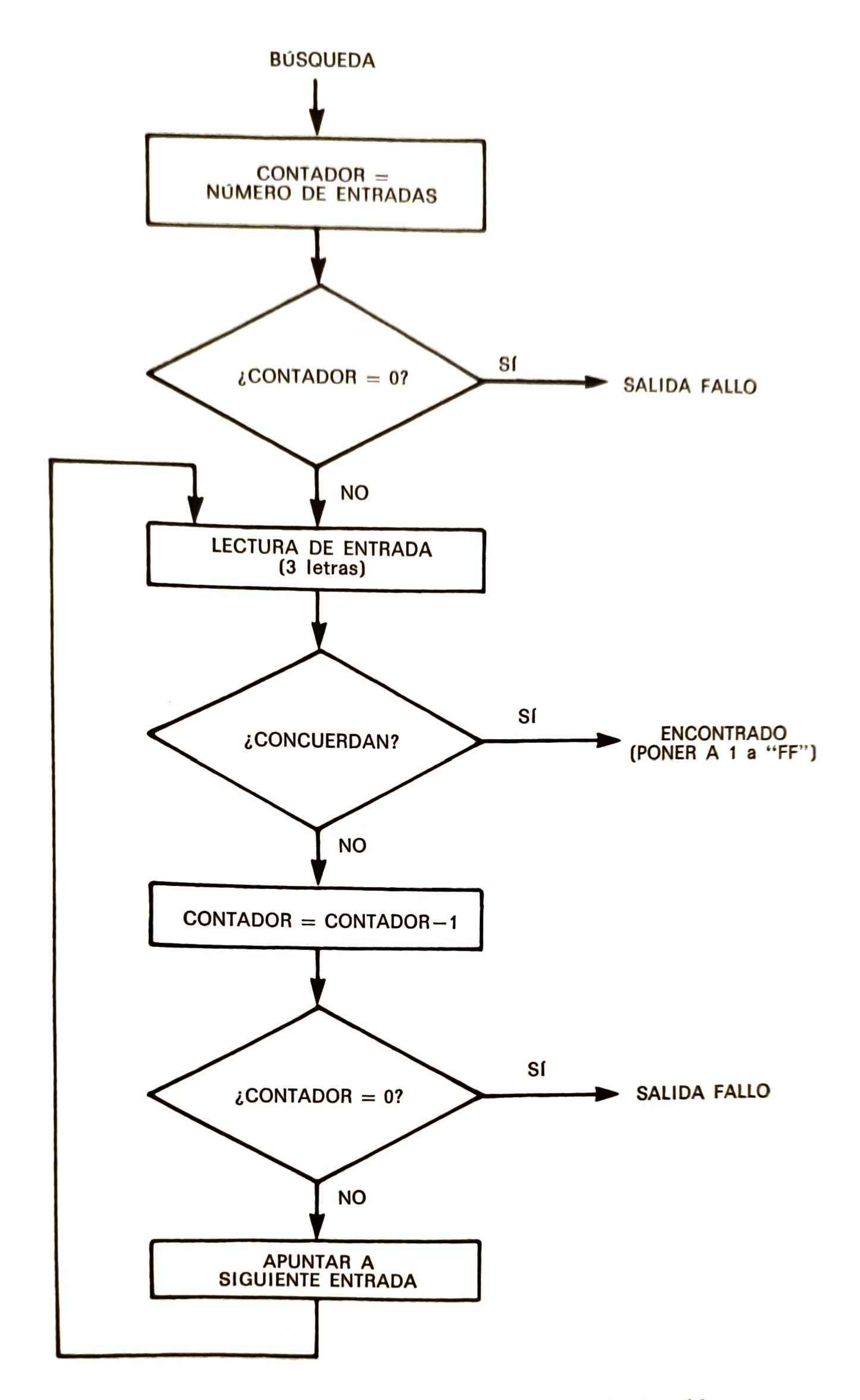


Figura 9-12 Diagrama de flujo de búsqueda de tablas.

El programa verifica, en primer lugar, que la nueva entrada no está ya en la lista (todas las etiquetas se supone que deben ser distintas en este ejemplo). Si no es así, se incrementa la longitud de la lista TABLEN y se desplaza OBJECT al final de la lista. En la figura 9-13 se muestra el diagrama de flujo correspondiente.

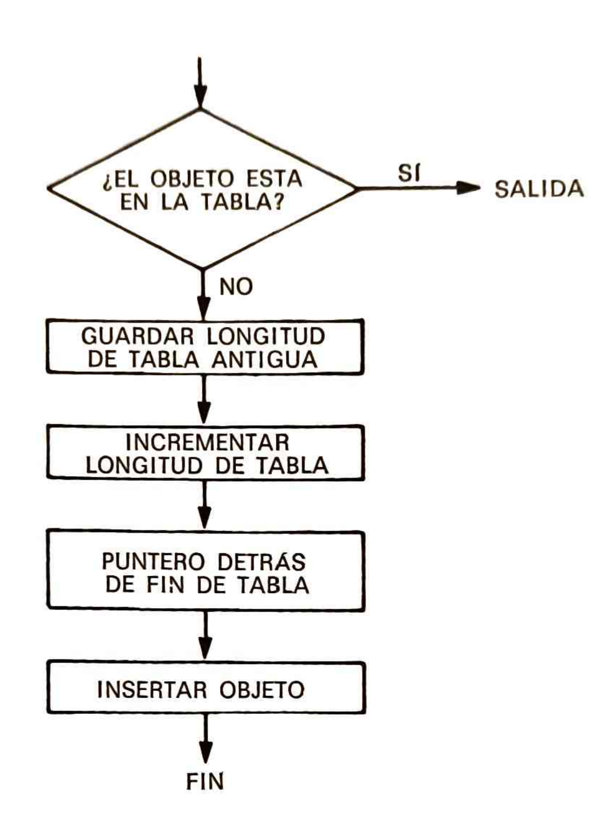


Figura 9-13 Diagrama de flujo de inserción de tablas.

El programa se muestra en la figura 9-16 al final de esta sección. Se llama "NEW" y reside en las posiciones de memoria 0636 a 0659.

Supresión de un elemento

Para suprimir un elemento de la lista, los elementos que le siguen en las direcciones más altas se desplazan sólo hacia arriba en una posición de un elemento. La longitud de la lista se disminuye. Esto se ilustra en la figura 9-14.

El programa correspondiente no presenta dificultad y aparece en la figura 9-16. Se denomina "DELETE" y reside en las direcciones de memoria 0659 a 0686. El diagrama de flujo se muestra en la figura 9-15.

La posición de memoria TEMPTR se utiliza como un puntero temporal que apunta al elemento que asciende.

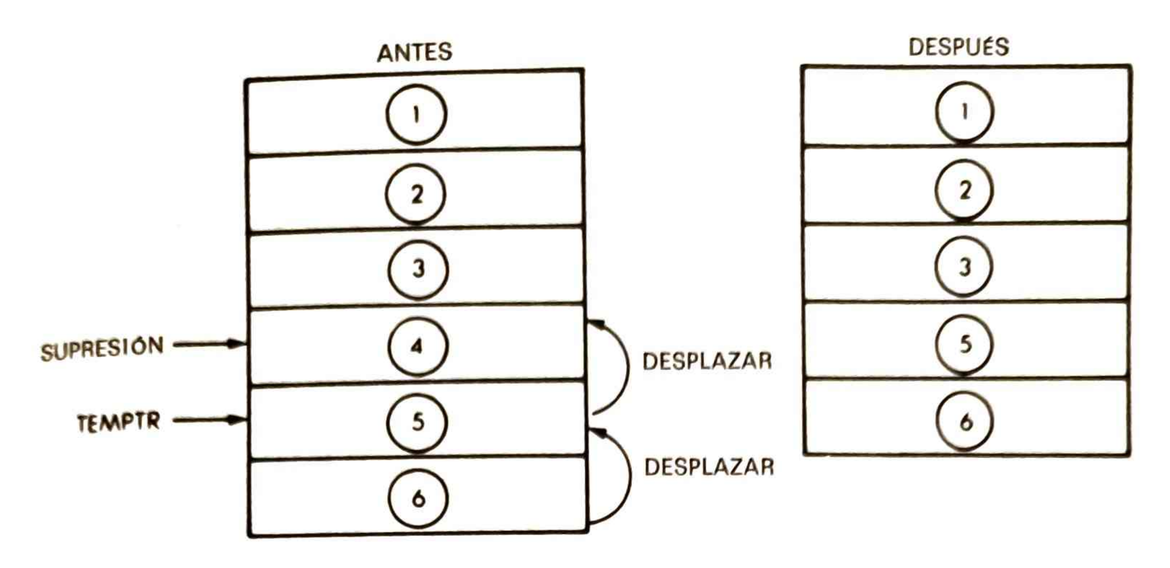


Figura 9-14 Supresión de una entrada (lista simple).

El registro índice Y se ajusta a la longitud de un elemento de la lista y se utiliza para transferir automáticamente a bloques. Obsérvese que se utiliza el direccionamiento indexado indirecto:

```
(0672)    LOOPE    DEY
                   LDA    (TEMPTR), Y
                   STA    (POINTR), Y
                   CPY    #0
                   BNE    LOOPE
```

Durante la transferencia, POINTR apunta siempre hacia el "hueco" de la lista, es decir, hacia el destino de la siguiente transferencia de bloque.

El indicador Z sirve para indicar una supresión satisfactoria a la salida.

LISTA ALFABÉTICA

Al contrario que la estructura anterior, la lista alfabética o "tabla" conserva todos los elementos clasificados en orden alfabético, lo que permite la utilización de técnicas de búsqueda más rápidas que la lineal. Se empleará en este caso una búsqueda binaria.

Búsqueda

El algoritmo de búsqueda es el de una búsqueda binaria clásica. Recordemos que la técnica es análoga, en lo esencial, a la utilizada para encontrar

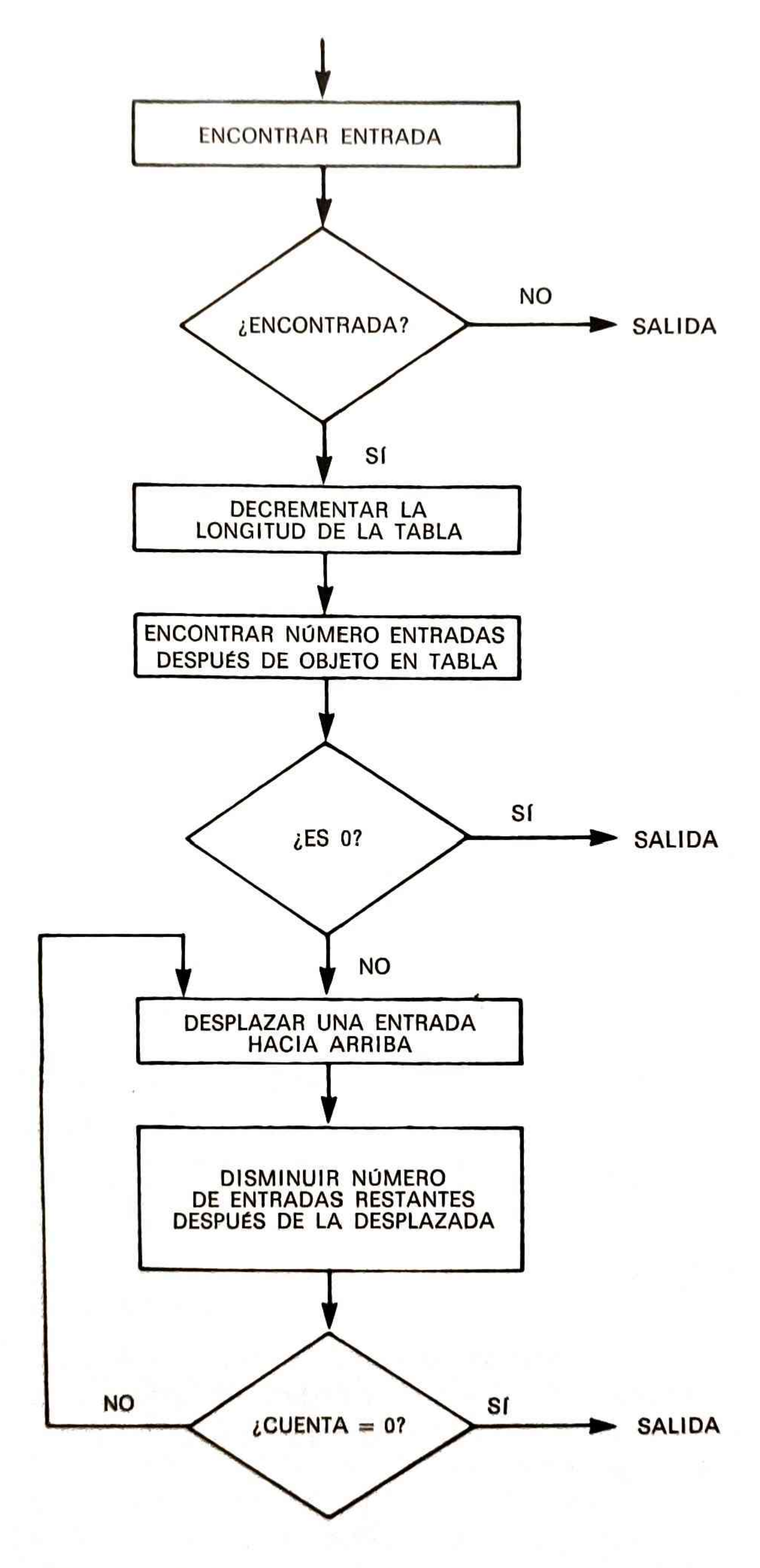


Figura 9-15 Diagrama de flujo de supresión de tabla.

```
LOC    CODE       LINE

0000              TABASE = $10
0000              POINTR = $12
0000              TABLEN = $14
0000              OBJECT = $15
0000              ENTLEN = $17
0000              TEMPTR = $18
0000              ;
0000                     *=$600
0600              ;
0600   A5 10      SEARCH LDA TABASE     ;INITIALIZE POINTER
0602   85 12             STA POINTR
0604   A5 11             LDA TABASE+1
0606   85 13             STA POINTR+1
0608   A6 14             LDX TABLEN     ;STORE TABLEN AS A VARIABLE
060A   F0 29             BEQ OUT        ;CHECK FOR 0 TABLE
060C   A0 00      ENTRY  LDY #0         ;COMPARE FIRST LETTERS
060E   B1 15             LDA (OBJECT),Y
0610   D1 12             CMP (POINTR),Y
0612   D0 0E             BNE NOGOOD
0614   C8                INY            ;COMPARE SECOND LETTERS
0615   B1 15             LDA (OBJECT),Y
0617   D1 12             CMP (POINTR),Y
0619   D0 07             BNE NOGOOD
061B   C8                INY            ;COMPARE THIRD LETTERS
061C   B1 15             LDA (OBJECT),Y
061E   D1 12             CMP (POINTR),Y
0620   F0 11             BEQ FOUND
0622   CA         NOGOOD DEX            ;SEE HOW MANY ENTRIES ARE LEFT
0623   F0 10             BEQ OUT
0625   A5 17             LDA ENTLEN     ;ADD ENTLEN TO POINTER
0627   18                CLC
0628   65 12             ADC POINTR
062A   85 12             STA POINTR
062C   90 DE             BCC ENTRY
062E   E6 13             INC POINTR+1
0630   4C 0C 06          JMP ENTRY
0633   A9 FF      FOUND  LDA #$FF       ;CLEAR Z FLAG IF FOUND
0635   60         OUT    RTS
0636              ;
0636              ;
0636              ;
0636   20 00 06   NEW    JSR SEARCH     ;SEE IF OBJECT IS THERE
0639   D0 1D             BNE OUTE
063B   A6 14             LDX TABLEN     ;CHECK FOR 0 TABLE
063D   F0 0D             BEQ INSERT
063F   A5 12             LDA POINTR     ;POINTER IS AT LAST ENTRY
0641   18                CLC            ;..MUST MOVE IT TO END OF TABLE
0642   65 17             ADC ENTLEN
0644   85 12             STA POINTR
0646   90 02             BCC INSERT
0648   E6 13             INC POINTR+1
064A   E6 14      INSERT INC TABLEN     ;INCREMENT TABLE LENGTH
064C   A0 00             LDY #0         ;MOVE OBJECT TO END OF TABLE
064E   A6 17             LDX ENTLEN
0650   B1 15      LOOP   LDA (OBJECT),Y
0652   91 12             STA (POINTR),Y
0654   C8                INY
0655   CA                DEX
0656   D0 F8             BNE LOOP
0658   60         OUTE   RTS            ;Z SET IF WAS DONE
0659              ;
0659              ;
0659              ;
0659   20 00 06   DELETE JSR SEARCH     ;FIND WHERE OBJECT IS
065C   F0 2B             BEQ OUTS       ;EXIT IF NOT FOUND
065E   C6 14             DEC TABLEN     ;DECREMENT TABLE LENGTH
0660   CA                DEX            ;SEE HOW MANY ENTRIES ARE
```

Figura 9-16   Programas de listas simples: búsqueda, introducción y supresión.

```
0661  F0 26            BEQ DONE        ;..AFTER ONE TO BE DELETED
0663  A5 12     ADDEN  LDA POINTR      ;ADD ENTLEN TO POINTER AND
0665  18               CLC             ;..SAVE AT TEMP STORAGE
0666  65 17            ADC ENTLEN
0668  85 18            STA TEMPTR
066A  A9 00            LDA #0
066C  65 13            ADC POINTR+1    ;ADD CARRY TO HIGH BYTE
066E  85 19            STA TEMPTR+1
0670  A4 17            LDY ENTLEN
0672  88        LOOPE  DEY
0673  B1 18            LDA (TEMPTR),Y  ;SHIFT ONE ENTRY OF MEMORY DOWN
0675  91 12            STA (POINTR),Y
0677  C0 00            CPY #0
0679  D0 F7            BNE LOOPE
067B  CA               DEX             ;DECREMENT ENTRY COUNTER
067C  F0 0B            BEQ DONE
067E  A5 18            LDA TEMPTR      ;MOVE TEMP TO POINTER
0680  85 12            STA POINTR
0682  A5 19            LDA TEMPTR+1
0684  85 13            STA POINTR+1
0686  4C 63 06         JMP ADDEN
0689  A9 FF     DONE   LDA #$FF        ;CLEAR Z FLAG IF IT WAS DONE
068B  60        OUTS   RTS
068C                   ;
068C                   ;
068C                   .END

ERRORS = 0000 <0000>


SYMBOL TABLE

SYMBOL   VALUE

ADDEN   0663   DELETE  0659   DONE    0689   ENTLEN  0017
ENTRY   060C   FOUND   0633   INSERT  064A   LOOP    0650
LOOPE   0672   NEW     0636   NOGOOD  0622   OBJECT  0015
OUT     0635   OUTE    0658   OUTS    068B   POINTR  0012
SEARCH  0600   TABASE  0010   TABLEN  0014   TEMPTR  0018

END OF ASSEMBLY
```

Figura 9-16 (Continuación).

un nombre en una guía telefónica. Se suele comenzar en cualquier parte en medio del libro y a continuación, dependiendo de los nombres encontrados, se va hacia adelante o bien hacia atrás para encontrar el nombre deseado. Este método es rápido y sencillo de realizar.

El diagrama de flujo de búsqueda binaria se muestra en la figura 9-17 y el programa se muestra en la figura 9-22.

La lista conserva las entradas en orden alfabético y se accede a ella con el empleo de una lógica binaria o "logarítmica". Un ejemplo se muestra en la figura 9-18.

La búsqueda es un poco complicada por la necesidad de conservar el seguimiento de varias condiciones. El problema principal que se ha de evitar es la búsqueda de un objeto que falte en la lista, ya que, en este caso, estaríamos

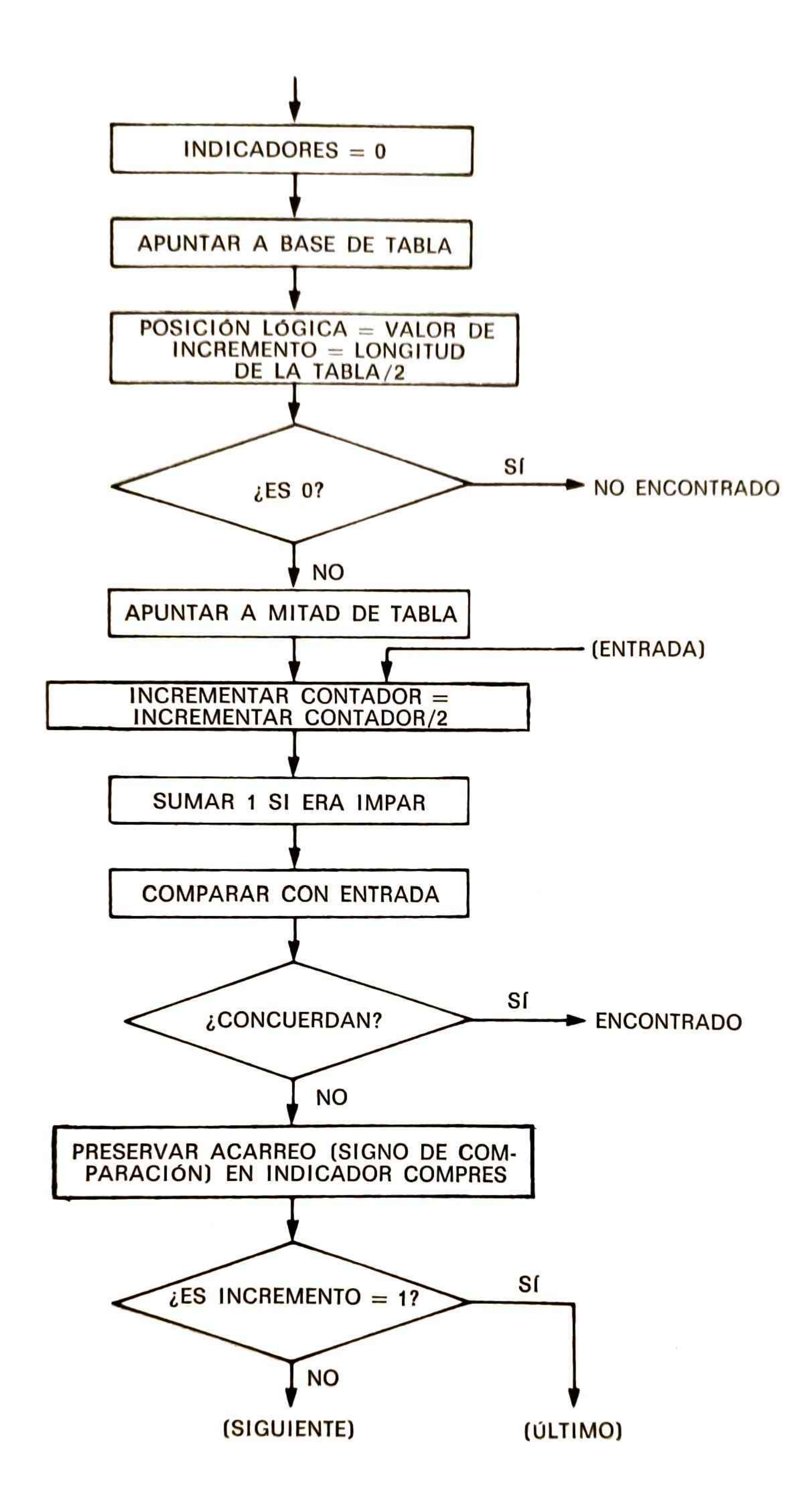


Figura 9-17 Diagrama de flujo de búsqueda binaria.

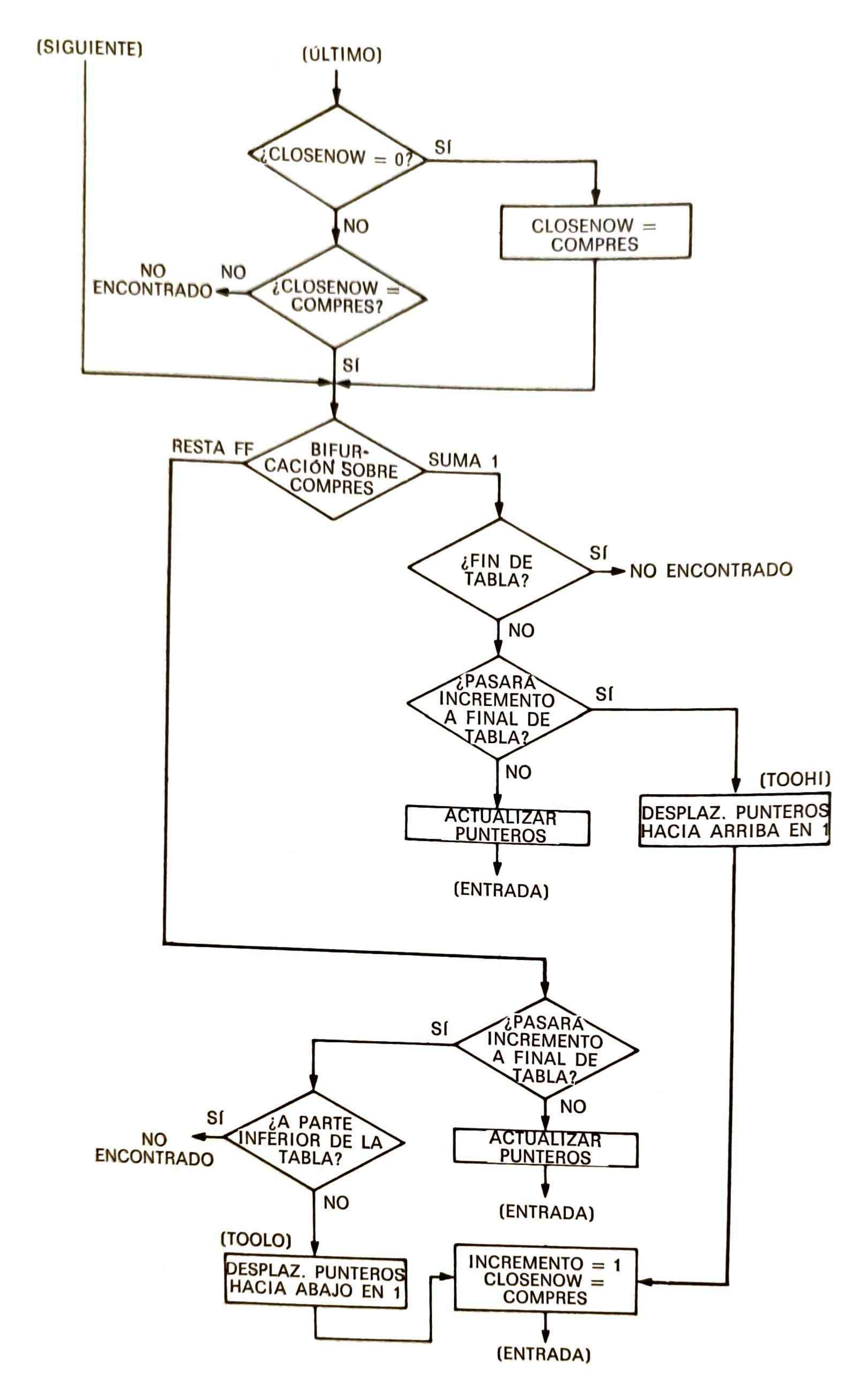


Figura 9-17 (Continuación).

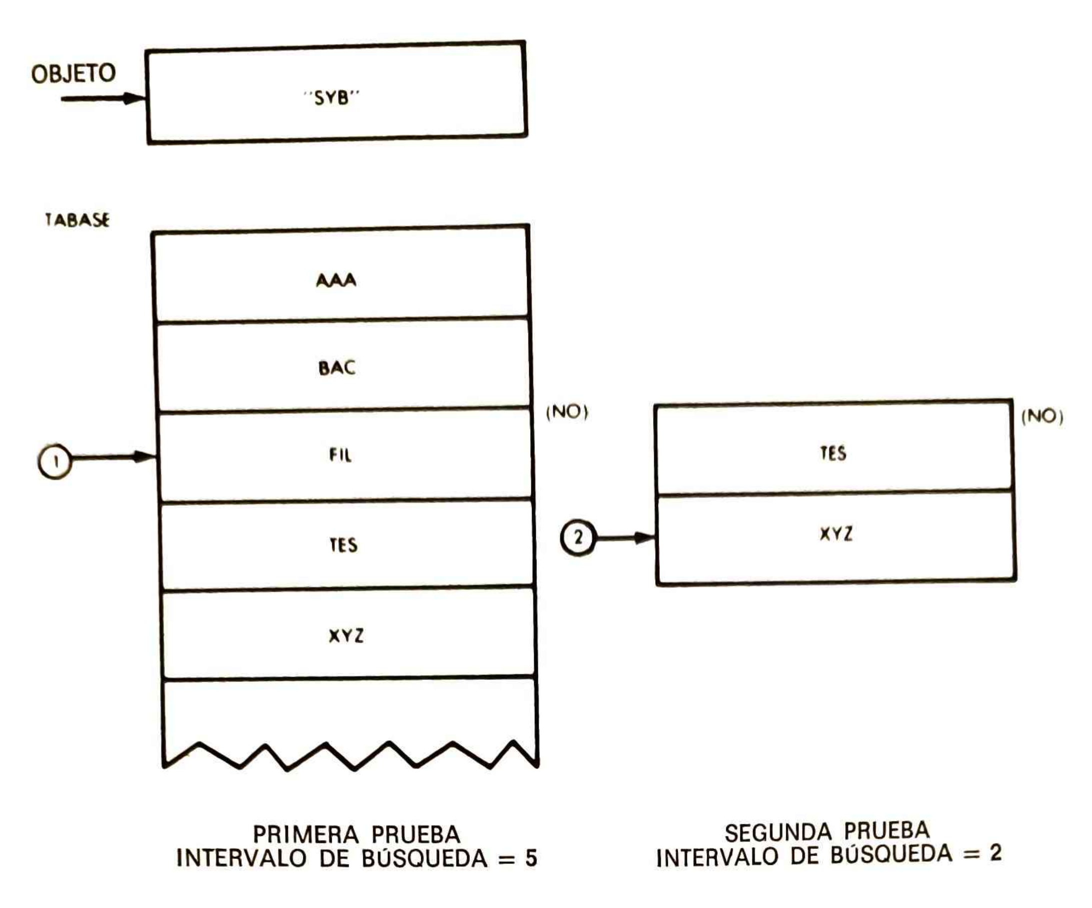


Figura 9-18 Una búsqueda binaria.

comprobando siempre, alternativamente, los valores alfabético inmediatamente superior e inferior. Para evitarlo, se mantiene un indicador en el programa que conserva el valor del indicador de acarreo después de una comparación no satisfactoria. Cuando el valor INCMNT, que indica en cuánto se incrementará el puntero a continuación, alcanza un valor de "1", otro indicador llamado "CLOSE" se pone al valor del indicador CMPRES. En consecuencia, ya que los incrementos posteriores serán "1", si el puntero sobrepasa el punto en donde debiera estar el objeto, CMPRES no será igual a CLOSE y la búsqueda terminará. Esta característica activa también a la rutina NEW para determinar dónde están localizados los punteros físico y lógico, relativos a la posición a donde irá el objeto.

Así, si el objeto buscado no está en la tabla y el puntero móvil se incrementa en uno, el indicador CLOSE se posicionará. Después del siguiente paso de la rutina, el resultado de la comparación será opuesto al anterior. Los dos indicadores no concordarán y el programa saldrá indicando "no encontrado".

El otro problema importante que se debe tratar es la posibilidad de salir de la tabla cuando se suma o resta el valor del incremento. Se soluciona realizando una prueba de "suma" o de "resta" utilizando el puntero lógico y el valor de su longitud para determinar el número efectivo de entradas, en lugar

de utilizar punteros físicos para determinar simplemente sus posiciones físicas.

En resumen, se emplean dos indicadores por el programa para memorizar información: CMPRES y CLOSE. El indicador CMPRES sirve para memorizar el hecho de que el acarreo era "0" o "1" después de la comparación más reciente. Ello determina si el elemento bajo prueba era más grande o más pequeño que el que se estaba comparando. Siempre que el acarreo C es "1", la entrada es más pequeña que el objeto buscado y CMPRES se pone a "1". Si el acarreo C es "0", la entrada es más grande que el objeto y CMPRES se pondrá a "FF".

Nótese también que cuando el acarreo está en "1", el puntero móvil apuntará hacia el elemento situado por debajo del objeto.

El segundo indicador utilizado por el programa es CLOSE. Este indicador es igual a CMPRES si el incremento de búsqueda INCMNT se hace igual a "1". Detectará el hecho de que el elemento no se ha encontrado si CMPRES no es igual a CLOSE en la siguiente vuelta.

Otras variables utilizadas por el programa son:

LOGPOS, que indica la posición lógica en la tabla (número del elemento).

INCMNT, que representa el valor cuyo puntero móvil se incrementará o decrementará si no da igualdad la siguiente comparación.

TABLEN representa, habitualmente, la longitud de la lista. LOGPOS e INCMNT son comparadas con TABLEN para asegurarse de que los límites de la lista no son superados.

El programa denominado "SEARCH" se muestra en la figura 9-22. Reside en las posiciones de memoria 0600 a 06E3 y merece ser estudiado con cuidado, ya que es mucho más complejo que en el caso de una búsqueda lineal.

Una complicación adicional se debe al hecho de que el intervalo de búsqueda puede ser, a veces, par o impar. Si es par deberá introducirse una corrección. No se puede, por ejemplo, apuntar al elemento central de una lista de cuatro elementos.

Cuando es impar, se utiliza un "truco" para apuntar al elemento central: la división por 2 se realiza por desplazamiento a derecha. El bit que sale fuera al acarreo después de la instrucción LSR será "1" si el intervalo fue impar. Se suma de nuevo, simplemente al puntero:

```
(0615)   DIV   LSR   A        DIVIDIR POR DOS
               ADC   #0       CAPTAR ACARREO
               STA   LOGPOS   NUEVO PUNTERO
```

El objeto buscado se compara entonces con la entrada en medio del nuevo intervalo de búsqueda. Si la comparación da igualdad, el programa sale. En caso contrario ("NOGOOD"), el acarreo se pone a 0 si el objeto

es menor que la entrada. Cuando el INCMNT se hace "1", se verifica que el indicador CLOSE (que se ha inicializado a "0") estaba puesto a uno. En caso contrario, se pondrá en dicho estado, pero será preciso realizar una prueba para determinar si hemos sobrepasado la posición en que el objeto debía estar pero en la que no fue encontrado.

Inserción de un elemento

Para insertar un nuevo elemento, es preciso efectuar una búsqueda binaria. Si el elemento se encuentra en la tabla, no necesita ser insertado (supongamos en este caso que todos los elementos son distintos). Si el elemento no se encontró en la tabla, se debe insertar. El valor del indicador CMPRES, después de la búsqueda, indica si este elemento se debe insertar inmediatamente antes, o inmediatamente después, del último elemento con el que se comparó. Todos los elementos que siguen a la nueva posición en donde se va a situar se desplazan hacia abajo en una posición de bloque y se inserta el nuevo elemento.

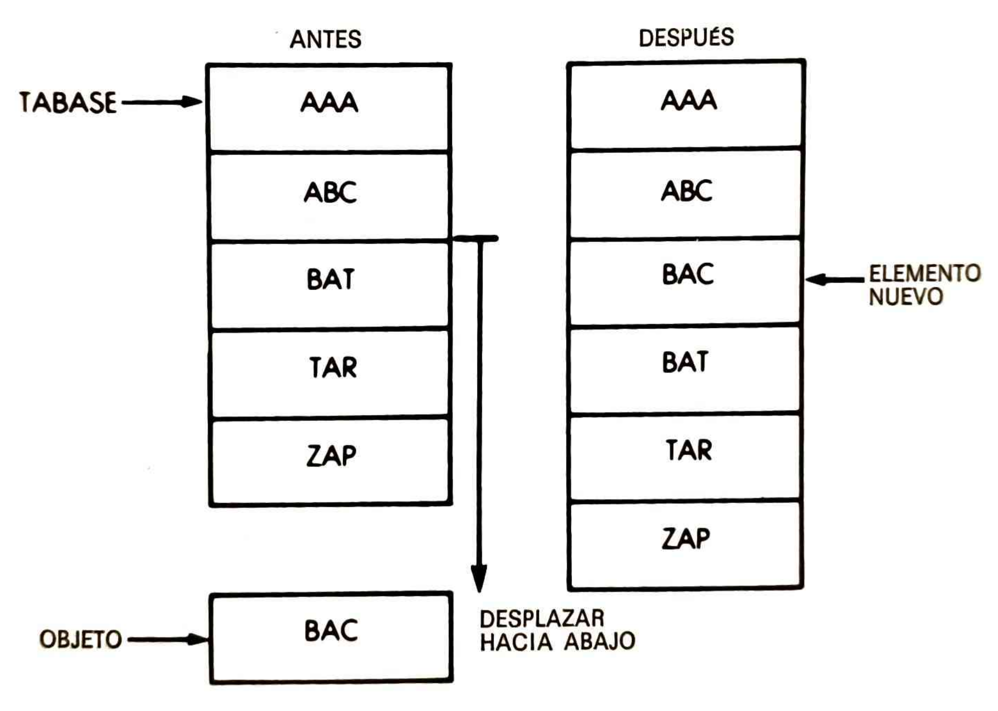


Figura 9-19 Insertar: "BAC".

El proceso de inserción se ilustra en la figura 9-19 y el programa correspondiente se muestra en la figura 9-22.

El programa se llama "NEW" y reside en las posiciones de memoria 06E3 a 075E.

Obsérvese que el direccionamiento indexado indirecto se utiliza, de nuevo, por transferencias de bloques:

```
(072A)           LDY   ENTLEN
        ANOTHR   DEY
                 LDA   (POINTR), Y
                 STA   (TEMP), Y
                 CPY   #0
                 BNE   ANOTHR
```

Obsérvese lo mismo en la posición de memoria 0750.

Supresión de un elemento

De modo similar, para suprimir un elemento es preciso efectuar una búsqueda binaria para encontrar el objeto. Si la búsqueda no es satisfactoria, no necesitará ser suprimido. Si la búsqueda es positiva, se suprime el elemento y todos los elementos siguientes se desplazan una posición de bloque. Un ejemplo representativo se muestra en la figura 9-20 y el programa en la figura 9-22. El diagrama de flujo se indica en la figura 9-21.

Se llama "DELETE" y reside en las direcciones de memoria 075F a 0799.

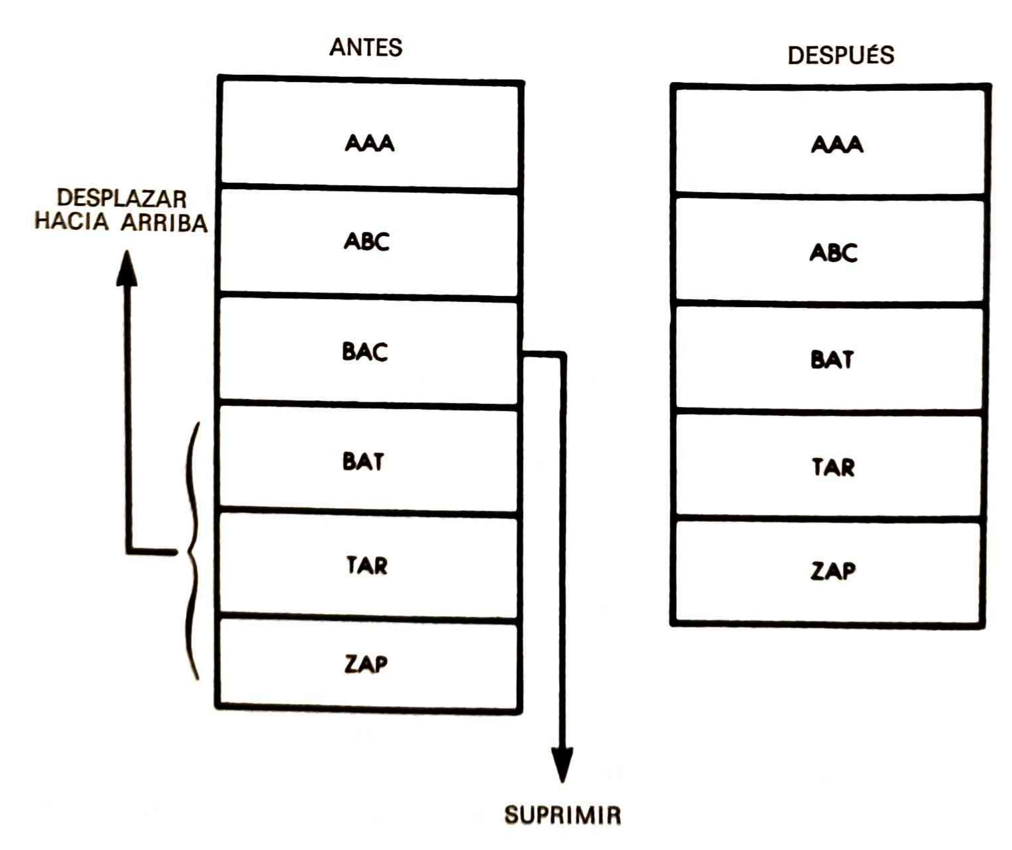


Figura 9-20 Suprimir: "BAC".

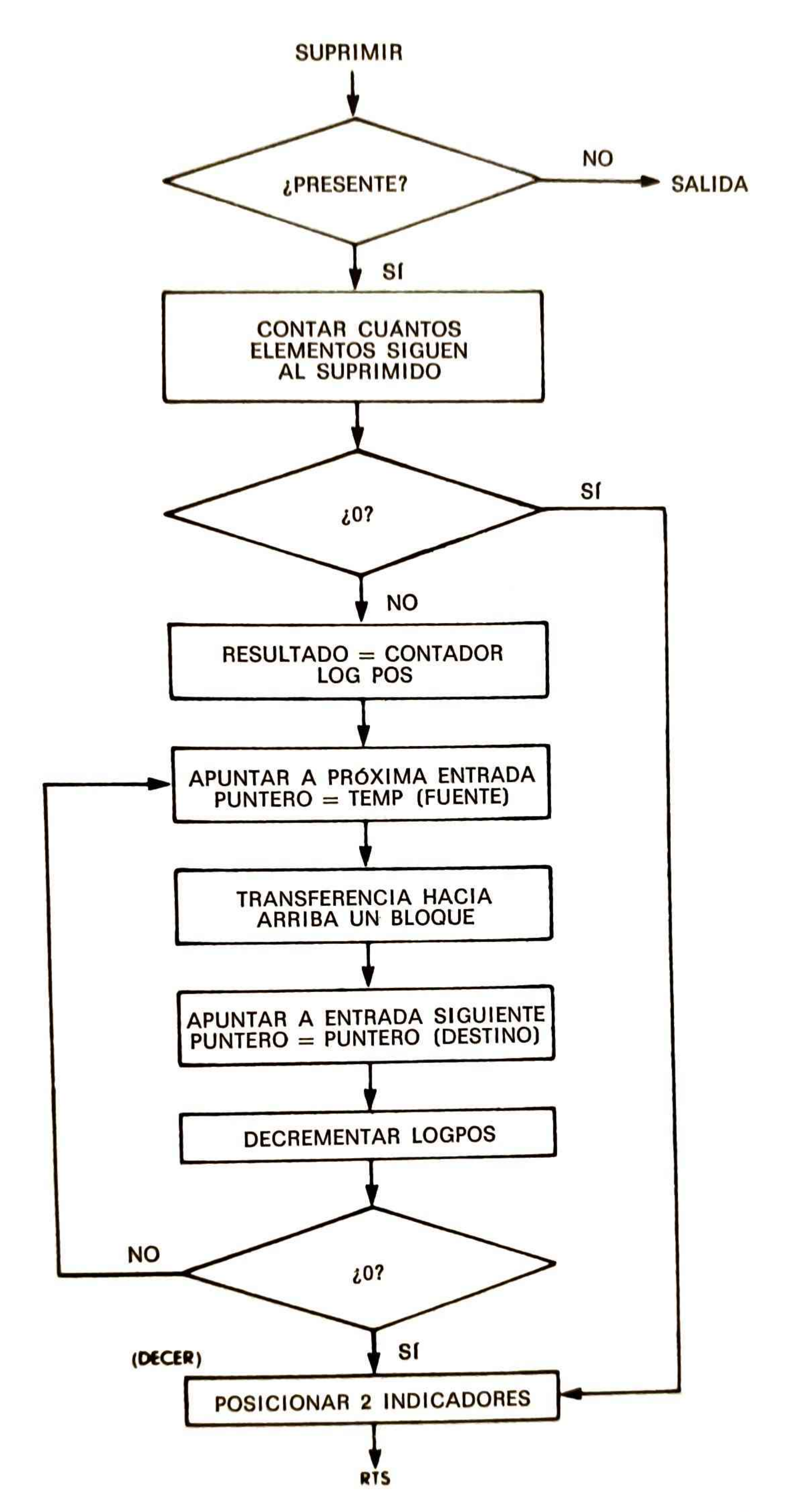


Figura 9-21 Diagrama de flujo de supresión (lista alfabética).

```
LOC     CODE       LINE

0000               CLOSE  = $10
0000               CMPRES = $11
0000               TABASE = $12
0000               POINTR = $14
0000               TABLEN = $16
0000               LOGPOS = $17
0000               INCMNT = $18
0000               TEMP   = $19
0000               ENTLEN = $1B
0000               OBJECT = $1C
0000               ;
0000                      * = $600
0600               ;
0600   A9 00       SEARCH LDA #0          ;ZERO FLAGS
0602   85 10              STA CLOSE
0604   85 11              STA CMPRES
0606   A5 12              LDA TABASE      ;INITIALIZE POINTER
0608   85 14              STA POINTR
060A   A5 13              LDA TABASE+1
060C   85 15              STA POINTR+1
060E   A5 16              LDA TABLEN      ;GET TABLE LENGTH
0610   D0 03              BNE DIV
0612   4C E0 06           JMP OUT
0615   4A          DIV    LSR A           ;DIVIDE IT BY 2
0616   69 00              ADC #0          ;ADD BACK IN 1'S BIT
0618   85 17              STA LOGPOS      ;STORE AS LOGICAL POSITION
061A   85 18              STA INCMNT      ;STORE AS INCREMENT VALUE
061C   A6 17              LDX LOGPOS      ;MULTIPLY ENTLEN BY LOGPOS
061E   CA                 DEX             ;..ADDING RESULT TO POINTER
061F   F0 0E              BEQ ENTRY
0621   A5 1B       LOOP   LDA ENTLEN
0623   18                 CLC
0624   65 14              ADC POINTR
0626   85 14              STA POINTR
0628   90 02              BCC LOPP
062A   E6 15              INC POINTR+1
062C   CA          LOPP   DEX
062D   D0 F2              BNE LOOP
062F   A5 18       ENTRY  LDA INCMNT      ;DIVIDE INCREMENT VALUE BY 2
0631   4A                 LSR A
0632   69 00              ADC #0
0634   85 18              STA INCMNT
0636   A0 00              LDY #0          ;COMPARE FIRST LETTERS
0638   B1 1C              LDA (OBJECT),Y
063A   D1 14              CMP (POINTR),Y
063C   D0 11              BNE NOGOOD
063E   C8                 INY             ;COMPARE 2ND LETTERS
063F   B1 1C              LDA (OBJECT),Y
0641   D1 14              CMP (POINTR),Y
0643   D0 0A              BNE NOGOOD
0645   C8                 INY             ;COMPARE 3RD LETTERS
0646   B1 1C              LDA (OBJECT),Y
0648   D1 14              CMP (POINTR),Y
064A   D0 03              BNE NOGOOD
064C   4C E2 06           JMP FOUND
064F   A0 FF       NOGOOD LDY #$FF        ;SET COMPARE RESULT FLAG
0651   90 02              BCC TESTS       ;IF OBJ < POINTR : C=0
0653   A0 01              LDY #1
0655   84 11       TESTS  STY CMPRES
0657   A4 18              LDY INCMNT      ;IS INCR. VALUE A 1?
0659   88                 DEY
065A   D0 10              BNE NEXT
065C   A5 10              LDA CLOSE       ;CHECK CLOSE FLAG IF IT WAS
065E   F0 08              BEQ MAKCLO      ;IF CLOSE FLAG NOT SET, GO DO IT
0660   38                 SEC
0661   E5 11              SBC CMPRES      ;SEE IF GAVE PASSED WHERE OBJ.
0663   F0 07              BEQ NEXT        ;..SHOULD BE BUT ISNT
```

Figura 9-22 Programas de lista alfabética: búsqueda binaria, supresión, inserción.

```
0665  4C E0 06          JMP OUT
0668  A5 11     MAKCLO  LDA CMPRES      ;SET CLOSE FLAG TO CMPRES
066A  85 10             STA CLOSE
066C  24 11     NEXT    BIT CMPRES
066E  30 35             BMI SUBIT
0670  A5 16             LDA TABLEN      ;SEE IF ADDITIION OF INCMNT
0672  38                SEC             ;..WILL RUN PAST END OF TABLE
0673  E5 17             SBC LOGPOS
0675  F0 69             BEQ OUT         ;CHECK TO SEE IF AT END OF TABLE ALREADY
0677  E5 18             SBC INCMNT
0679  90 1A             BCC TOOHI
067B  A6 18             LDX INCMNT      ;IS ALL RIGHT, INC POINTER BY
067D  A5 1B     ADDER   LDA ENTLEN      ;..PROPER AMOUNT
067F  18                CLC
0680  65 14             ADC POINTR
0682  85 14             STA POINTR
0684  90 02             BCC AD1
0686  E6 15             INC POINTR+1
0688  CA        AD1     DEX
0689  D0 F2             BNE ADDER
068B  A5 17             LDA LOGPOS      ;INCREMENT LOGICAL POSITION
068D  18                CLC
068E  65 18             ADC INCMNT
0690  85 17             STA LOGPOS
0692  4C 2F 06          JMP ENTRY
0695  E6 17     TOOHI   INC LOGPOS      ;INCR. LOGICAL POSITION
0697  A5 1B             LDA ENTLEN      ;MOVE POINTER UP ONE ENTRY
0699  18                CLC
069A  65 14             ADC POINTR
069C  85 14             STA POINTR
069E  90 35             BCC SETCLO
06A0  E6 15             INC POINTR+1
06A2  4C D5 06          JMP SETCLO
06A5  A5 17     SUBIT   LDA LOGPOS      ;SEE IF INC WILL GO OFF BOTTOM
06A7  38                SEC             ;.. OF TABLE
06A8  E5 18             SBC INCMNT
06AA  F0 17             BEQ TOOLOW
06AC  90 15             BCC TOOLOW
06AE  85 17             STA LOGPOS      ;SAVE NEW LOGICAL POSITION
06B0  A6 18             LDX INCMNT
06B2  A5 14     SUBLOP  LDA POINTR      ;SUBTRACT PROPER AMT. FROM POINTER
06B4  38                SEC
06B5  E5 1B             SBC ENTLEN
06B7  85 14             STA POINTR
06B9  B0 02             BCS SUBO
06BB  C6 15             DEC POINTR+1
06BD  CA        SUBO    DEX
06BE  D0 F2             BNE SUBLOP
06C0  4C 2F 06          JMP ENTRY
06C3  A6 17     TOOLOW  LDX LOGPOS      ;SEE IF POS IS ALREADY 1
06C5  CA                DEX
06C6  F0 18             BEQ OUT
06C8  C6 17             DEC LOGPOS
06CA  A5 14             LDA POINTR      ;SUB 1 ENTRY FROM POINTER
06CC  38                SEC
06CD  E5 1B             SBC ENTLEN
06CF  85 14             STA POINTR
06D1  B0 02             BCS SETCLO
06D3  C6 15             DEC POINTR+1
06D5  A9 01     SETCLO  LDA #1
06D7  85 18             STA INCMNT
06D9  A5 11             LDA CMPRES
06DB  85 10             STA CLOSE
06DD  4C 2F 06          JMP ENTRY
06E0  A2 FF     OUT     LDX #$FF        ;Z SET IF FOUND
06E2  60        FOUND   RTS
06E3            ;
06E3            ;
06E3            ;
06E3  20 00 06  NEW     JSR SEARCH      ;SEE IF OBJECT IS ALREADY THERE
```

Figura 9-22 (Continuación).

```
06E6  F0 76              BEQ OUTE
06E8  A5 16              LDA TABLEN       ;CHECK FOR 0 TABLE
06EA  F0 62              BEQ INSERT
06EC  24 11              BIT CMPRES       ;TEST LAST COMPARE RESULT
06EE  10 05              BPL LOSIDE
06F0  C6 17              DEC LOGPOS       ;SET LOGICAL POSITION SO
06F2  4C 00 07           JMP SETUP        ;..SUB WORKS LATER
06F5  A5 1B     LOSIDE   LDA ENTLEN       ;SET POINTER ABOVE WHERE
06F7  18                 CLC              ;..OBJECT WILL GO
06F8  65 14              ADC POINTR
06FA  85 14              STA POINTR
06FC  90 02              BCC SETUP
06FE  E6 15              INC POINTR+1
0700  A5 16     SETUP    LDA TABLEN       ;SEE HOW MANY ENTRIES THERE
0702  38                 SEC              ;..ARE AFTER WHERE OBJ. WILL GO
0703  E5 17              SBC LOGPOS
0705  F0 47              BEQ INSERT
0707  AA                 TAX
0708  A8                 TAY
0709  88                 DEY              ;SEE IF ALREADY POINTING TO
070A  F0 0E              BEQ SETEMP       ;..LAST ENTRY
070C  A5 1B     UPLOOP   LDA ENTLEN       ;MOVE POINTER TO LAST ENTRY
070E  18                 CLC
070F  65 14              ADC POINTR
0711  85 14              STA POINTR
0713  90 02              BCC SET0
0715  E6 15              INC POINTR+1
0717  88        SET0     DEY
0718  D0 F2              BNE UPLOOP
071A  A5 14     SETEMP   LDA POINTR       ;ADD ENTLEN TO POINTER
071C  18                 CLC              ;..STORE AT TEMP
071D  65 1B              ADC ENTLEN
071F  85 19              STA TEMP
0721  90 01              BCC SET1
0723  C8                 INY              ;Y WAS ALREADY 0
0724  98        SET1     TYA
0725  18                 CLC
0726  65 15              ADC POINTR+1
0728  85 1A              STA TEMP+1
072A  A4 1B     MOVER    LDY ENTLEN       ;SET Y FOR SHIFT
072C  88        ANOTHR   DEY
072D  B1 14              LDA (POINTR),Y   ;MOVE A BYTE
072F  91 19              STA (TEMP),Y
0731  C0 00              CPY #0
0733  D0 F7              BNE ANOTHR
0735  A5 14              LDA POINTR       ;DECR. POINTER AND TEMP
0737  38                 SEC              ;..BY ENTLEN
0738  E5 1B              SBC ENTLEN
073A  85 14              STA POINTR
073C  B0 02              BCS M1
073E  C6 15              DEC POINTR+1
0740  CA        M1       DEX
0741  D0 D7              BNE SETEMP
0743  A5 1B              LDA ENTLEN       ;MOVE POINTER BACK TO
0745  18                 CLC              ;WHERE OBJ. WILL GO
0746  65 14              ADC POINTR
0748  85 14              STA POINTR
074A  90 02              BCC INSERT
074C  E6 15              INC POINTR+1
074E  A0 00     INSERT   LDY #0           ;MOVE OBJECT INTO TABLE
0750  A6 1B              LDX ENTLEN
0752  B1 1C     INNER    LDA (OBJECT),Y
0754  91 14              STA (POINTR),Y
0756  C8                 INY
0757  CA                 DEX
0758  D0 F8              BNE INNER
075A  E6 16              INC TABLEN       ;INCREMENT TABLE LENGTH
075C  A2 FF              LDX #$FF
```

Figura 9-22 (Continuación).

```
075E  60          OUTE    RTS              ;Z SET IF NOT DONE
075F                      ;
075F                      ;
075F                      ;
075F  20 00 06    DELETE  JSR SEARCH       ;GET ADDR OF OBJECT IN TABLE
0762  D0 35               BNE OUTS         ;SEE IF IT IS THERE
0764  A5 16               LDA TABLEN       ;SEE HOW MANY ENTRIES ARE
0766  38                  SEC              ;..LEFT AFTER OBJ. IN TABLE
0767  E5 17               SBC LOGPOS
0769  F0 2A               BEQ DECER
076B  85 17               STA LOGPOS       ;STORE RESULT AS A COUNTER
076D  A5 1B       BIGLOP  LDA ENTLEN       ;SET TEMP & ENTRY ABOVE 1 ENTRY ABOVE OBJ.
076F  18                  CLC
0770  65 14               ADC POINTR
0772  85 19               STA TEMP
0774  A9 00               LDA #0
0776  65 15               ADC POINTR+1
0778  85 1A               STA TEMP+1
077A  A6 1B               LDX ENTLEN       ;SET COUNTERS
077C  A0 00               LDY #0
077E  B1 19       BYTE    LDA (TEMP),Y     ;MOVE A BYTE
0780  91 14               STA (POINTR),Y
0782  C8                  INY              ;IS BLOCK MOVED YET?
0783  CA                  DEX
0784  D0 F8               BNE BYTE
0786  A5 1B               LDA ENTLEN
0788  18                  CLC
0789  65 14               ADC POINTR
078B  85 14               STA POINTR
078D  90 02               BCC D2
078F  E6 15               INC POINTR+1
0791  C6 17       D2      DEC LOGPOS
0793  D0 D8               BNE BIGLOP
0795  C6 16       DECER   DEC TABLEN
0797  A9 00               LDA #0           ;Z SET IF WAS DONE
0799  60          OUTS    RTS
079A                      .END

ERRORS = 0000 <0000>


SYMBOL TABLE

SYMBOL   VALUE

AD1      0688   ADDER    067D   ANOTHR   072C   BIGLOP   076D
BYTE     077E   CLOSE    0010   CMPRES   0011   D2       0791
DECER    0795   DELETE   075F   DIV      0615   ENTLEN   001B
ENTRY    062F   FOUND    06E2   INCMNT   0018   INNER    0752
INSERT   074E   LOGPOS   0017   LOOP     0621   LOPP     062C
LOSIDE   06F5   M1       0740   MAKCLO   0668   MOVER    072A
NEW      06E3   NEXT     066C   NOGOOD   064F   OBJECT   001C
OUT      06E0   OUTE     075E   OUTS     0799   POINTR   0014
SEARCH   0600   SETO     0717   SET1     0724   SETCLO   06B5
SETEMP   071A   SETUP    0700   SUBO     06BD   SUBIT    06A5
SUBLOP   06B2   TABASE   0012   TABLEN   0016   TEMP     0019
TESTS    0655   TOOHI    0695   TOOLOW   06C3   UPLOOP   070C

END OF ASSEMBLY
```

Figura 9-22 (Continuación).

LISTA ENLAZADA

Supongamos que la lista enlazada contiene, como es habitual, los tres caracteres alfanuméricos que sirven de etiqueta, seguidos de 1 a 250 bytes de datos, por un puntero de 2 bytes que contiene la dirección de comienzo del elemento siguiente y al final seguido por un marcador de 1 byte. Cuando este marcador de 1 byte se pone a "1", impide a la rutina de inserción sustituir por una entrada nueva la existente.

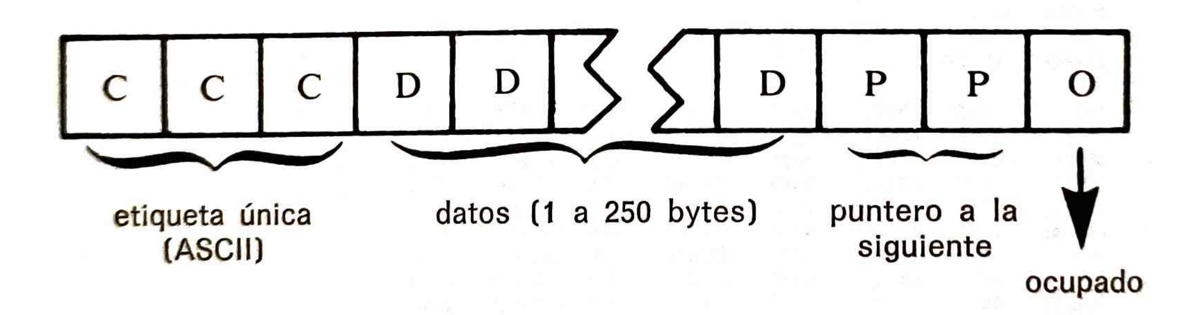
Además, un directorio contiene un puntero hacia el primer elemento de cada letra del alfabeto, para facilitar la recuperación. En el programa se supone que las etiquetas son caracteres alfabéticos ASCII. Todos los punteros, al final de la lista, se ponen al valor NIL, el cual se ha elegido igual a la base de la tabla ya que este valor no deberá encontrarse en la lista enlazada.

Los programas de inserción y supresión efectúan operaciones evidentes en los punteros. Utilizan el indicador INDEXD para indicar si un puntero que apunta a un objeto procedía de un elemento anterior en la lista o desde la tabla del directorio. Los programas correspondientes se muestran en la figura 9-27 y la estructura de datos se muestra en la figura 9-23.

Una aplicación para esta estructura de datos podría ser un libro de direcciones por ordenador, en donde cada persona se representa por un único código de 3 letras (por ejemplo, las iniciales) y el campo de datos contiene una dirección simplificada, más el número de teléfono (hasta 250 caracteres).

Examinemos más detalladamente la estructura en la figura 9-23.

El formato de un elemento, o entrada es:



Como de costumbre las notaciones son:

ENTLEN: longitud total de un elemento (en bytes)
TABASE: dirección de base de la lista
TABLEN: número de elementos (1 a 256).

Aquí REFBASE apunta a la dirección base del directorio, o "tabla de referencia".

Cada dirección de dos bytes en el interior de este directorio apunta a la

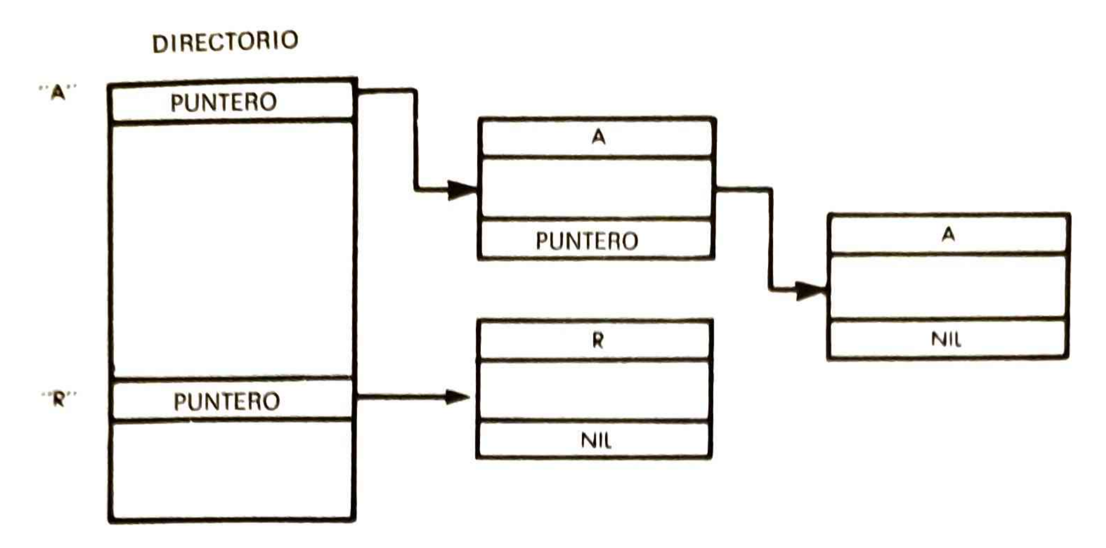


Figura 9-23 Estructura de listas enlazadas.

primera ocurrencia de la letra que le corresponde en la lista. Por tanto, cada grupo de elementos cuyas etiquetas comienzan por la misma letra forman una lista separada en el interior de la estructura completa. Esta característica facilita la búsqueda y es análoga a un libro de direcciones. Obsérvese que ningún dato se desplaza durante una inserción o supresión. Únicamente se cambian los punteros, como en toda estructura de lista enlazada bien concebida.

Si no se encuentra ningún elemento que comience con una letra específica, o si no hay ningún elemento que siga alfabéticamente al existente, los punteros apuntarán al principio de la tabla (="NIL"). En la parte inferior de la tabla, por convenio, se almacena un valor tal como el valor absoluto de la diferencia entre este valor y "Z", o sea, más grande que la diferencia entre "A" y "Z". Ello constituye una marca de fin de tabla (EOT). El valor EOT se supone que, en este caso, ocupa la misma cantidad de memoria que un elemento normal, pero, si se desea podría ocupar sólo un byte.

Las letras se suponen, en este caso, alfabéticas en código ASCII. Para que fuera de otro modo, sería preciso cambiar la constante de la rutina PRETAB.

El puntero de la marca fin de tabla se posiciona al valor del principio de la tabla ("NIL").

Por convenio, los punteros NIL, que se encuentran al final de una lista o en una posición del directorio que no apunta a una cadena, se ponen al valor de la base de tabla para proporcionar una identificación singular. Se podrá utilizar otro convenio. En particular, una marca diferente de EOT redundará en una economía de espacio, ya que ningún elemento NIL necesita conservarse para elementos inexistentes.

La inserción y borrado se efectúan del modo habitual (ver 1.ª parte de este capítulo) modificando simplemente los punteros requeridos. El indicador INDEXD sirve para señalar si el puntero hacia el objeto está en la tabla de referencia o en otro elemento de cadena.

Búsqueda

El programa SEARCH reside en las posiciones de memoria 0600 a 0650. Además, utiliza la subrutina PRETAB en la dirección 06F8.

El principio de la búsqueda es fácil:

1 — Tomar el elemento del directorio que corresponde a la primera letra de la etiqueta OBJECT (objeto).

2 — Tomar el puntero encontrado en el directorio. Acceder al elemento. Si se trata de NIL, el elemento no existe.

3 — Si no se trata de NIL, comparar el elemento con el objeto. Si concuerdan, la búsqueda ha resultado positiva. En caso contrario, tomar el puntero hacia la siguiente entrada hacia abajo de la lista.

4 — Volver a comenzar en 2.

Un ejemplo se muestra en la figura 9-24.

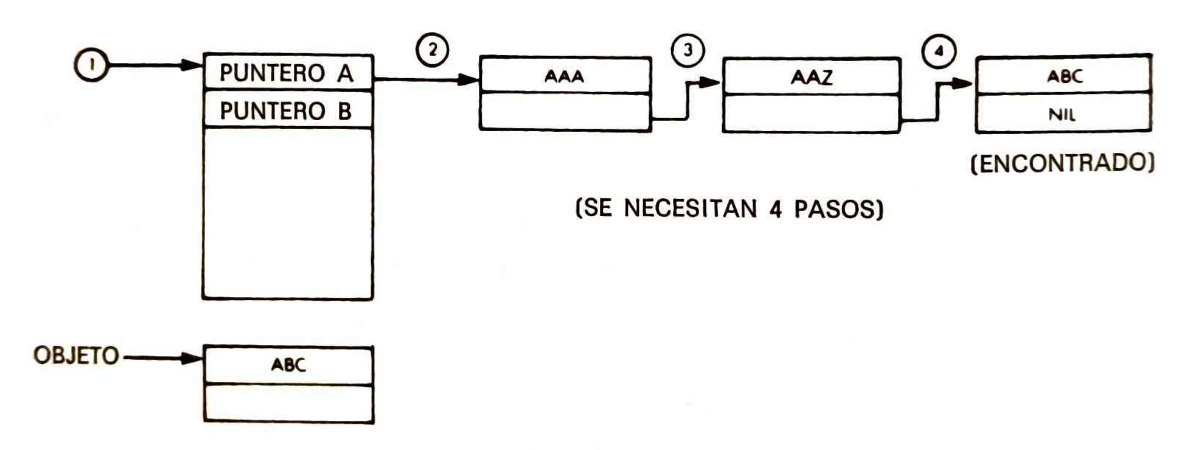


Figura 9-24 Lista enlazada: una búsqueda.

Inserción de un elemento

La inserción es esencialmente una búsqueda seguida por una inserción, una vez que se ha encontrado un "NIL". Se asigna al nuevo elemento un bloque de memoria más allá de la marca EOT buscando una marca de ocupación que se pone en posición "disponible". El programa se llama "NEW"

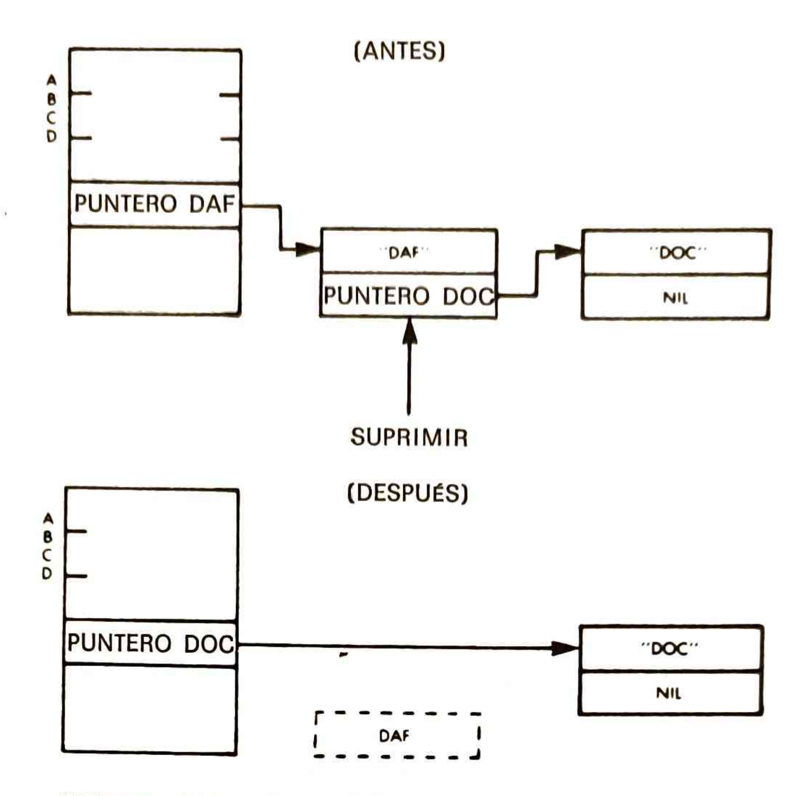


Figura 9-26 Ejemplo de supresión (lista enlazada).

enlace a su derecha es el "vástago a la derecha". Por ejemplo, la entrada para Jones contiene dos enlaces: "2" y "4". Esto indica que su vástago a la izquierda es el número de entrada 2 (Anderson) y su vástago a la derecha es el número de entrada 4 (Smith). Un "0" en el campo de enlace significa "sin descendencia". Una etiqueta de vástago a la izquierda precede alfabéticamente a su padre. Una etiqueta de vástago a la derecha le sucede.

Los dos programas principales para la gestión de un árbol son el de la *construcción del árbol* y el del *recorrido del árbol.* El elemento a insertar será situado en un buffer (memoria intermedia). El programa de construcción del árbol insertará el contenido del buffer en el árbol en el nudo adecuado. El programa de recorrido del árbol se dice que atraviesa el árbol recursivamente e imprime el contenido de cada uno de sus nudos en orden alfanumérico. El diagrama de flujo de la construcción del árbol se muestra en la figura 9-30 y el del recorrido del árbol se muestra en la figura 9-31.

En la medida en que la rutina de recorrido es recursiva, se presta mal a la representación del diagrama de flujo. Otra descripción de la rutina en un lenguaje de alto nivel se muestra, por tanto, en la figura 9-32. El formato exacto de un nudo o nodo del árbol se muestra en la figura 9-33. Contiene datos de longitud ENTLEN y luego dos punteros de 16 bits, el puntero dere-

```
LOC    CODE      LINE

0000             INDEXD = $10
0000             INDLOC = $11
0000             POINTR = $13
0000             OBJECT = $15
0000             TEMP   = $17
0000             REFBAS = $19
0000             OLD    = $1B
0000             TABASE = $1D
0000             ENTLEN = $1F
0000             ;
0000                    * = $600
0600             ;
0600  A9 01      SEARCH LDA #1           ;INITIALIZE FLAGS
0602  85 10             STA INDEXD
0604  20 F0 06          JSR PRETAD
0607  B1 11             LDA (INDLOC),Y   ;GET REF. POINTER FOR START
0609  85 13             STA POINTR       ;PUT IT IN POINTR
060B  C8                INY
060C  B1 11             LDA (INDLOC),Y
060E  85 14             STA POINTR+1
0610  A0 00      ENTRY  LDY #0           ;SEE IF ENTRY IS EOT VALUE
0612  B1 13             LDA (POINTR),Y
0614  C9 7C             CMP #$7C
0616  F0 36             BEQ NOTFND
0618  B1 15             LDA (OBJECT),Y   ;COMPARE FIRST LETTERS
061A  D1 13             CMP (POINTR),Y
061C  90 30             BCC NOTFND
061E  D0 12             BNE NOGOOD
0620  C8                INY              ;COMPARE SECOND LETTERS
0621  B1 15             LDA (OBJECT),Y
0623  D1 13             CMP (POINTR),Y
0625  90 27             BCC NOTFND
0627  D0 09             BNE NOGOOD
0629  C8                INY              ;COMPARE THIRD LETTERS
062A  B1 15             LDA (OBJECT),Y
062C  D1 13             CMP (POINTR),Y
062E  90 1E             BCC NOTFND
0630  F0 1E             BEQ FOUND
0632  A5 14      NOGOOD LDA POINTR+1     ;SAVE POINTR FOR POSSIBLE REF.
0634  85 1C             STA OLD+1
0636  A5 13             LDA POINTR
0638  85 1B             STA OLD
063A  A4 1F             LDY ENTLEN       ;GET POINTER FROM ENTRY AND
063C  B1 13             LDA (POINTR),Y   ;..LOAD IT INTO POINTR
063E  AA                TAX
063F  C8                INY
0640  B1 13             LDA (POINTR),Y
0642  85 14             STA POINTR+1
0644  8A                TXA
0645  85 13             STA POINTR
0647  A9 00             LDA #0
0649  85 10             STA INDEXD       ;RESET FLAG
064B  4C 10 06          JMP ENTRY
064E  A9 FF      NOTFND LDA #$FF
0650  60         FOUND  RTS              ;Z SET IF FOUND
0651             ;
0651             ;
0651             ;
0651  20 00 06   NEW    JSR SEARCH       ;SEE IF OBJ. IS ALREADY THERE
0654  F0 67             BEQ OUTE
0656  A5 1D             LDA TABASE       ;LOOK FOR UNOCCUPIED ENTRY
0658  18                CLC              ;..BLOCK
0659  69 01             ADC #1           ;JUMP PAST EOT VALUE
065B  85 17             STA TEMP
065D  A9 00             LDA #0
065F  65 1E             ADC TABASE+1
0661  85 18             STA TEMP+1
0663  A4 1F             LDY ENTLEN       ;SET Y TO POINT TO OCCUPANCY
```

Figura 9-27 Programa de lista enlazada.

```
0665  C8                  INY             ;..MARKER OF AN ENTRY
0666  C8                  INY
0667  A9 01       LOOP    LDA #1          ;TEST FOR OCCUPANCY MARKER
0669  D1 17               CMP (TEMP),Y
066B  D0 16               BNE INSERT
066D  A5 17               LDA TEMP        ;IF IS USED, MOVE TEMP TO NEXT
066F  18                  CLC             ;..ENTRY BLOCK
0670  65 1F               ADC ENTLEN
0672  90 02               BCC MORE
0674  E6 18               INC TEMP+1
0676  69 03       MORE    ADC #3
0678  85 17               STA TEMP
067A  A9 00               LDA #0
067C  65 18               ADC TEMP+1
067E  85 18               STA TEMP+1
0680  4C 67 06            JMP LOOP
0683  88          INSERT  DEY             ;SET Y BACK TO POINTING TO
0684  88                  DEY             ;..TOP OF DATA
0685  88          LOPE    DEY             ;MOVE OBJECT INTO SPACE
0686  B1 15               LDA (OBJECT),Y
0688  91 17               STA (TEMP),Y
068A  C0 00               CPY #0
068C  D0 F7               BNE LOPE
068E  A4 1F               LDY ENTLEN      ;PUT THE VALUE OF POINTR, THE
0690  A5 13               LDA POINTR      ;ENTRY AFTER OBJECT, INTO
0692  91 17               STA (TEMP),Y    ;POINTER AREA OF OBJECT
0694  C8                  INY
0695  A5 14               LDA POINTR+1
0697  91 17               STA (TEMP),Y
0699  C8                  INY
069A  A9 01               LDA #1          ;SET OCCUPANCY MARKER
069C  91 17               STA (TEMP),Y
069E  A5 10               LDA INDEXD      ;TEST TO SEE IF REF. TABLE
06A0  D0 0D               BNE SETINX      ;..NEEDS READJUSTING
06A2  88                  DEY
06A3  A5 18               LDA TEMP+1      ;NO, CHANGE PREVIOUS ENTRY'S
06A5  91 1B               STA (OLD),Y     ;..POINTER
06A7  88                  DEY
06A8  A5 17               LDA TEMP
06AA  91 1B               STA (OLD),Y
06AC  4C BB 06            JMP DONE
06AF  20 F6 06    SETINX  JSR PRETAB      ;GET ADDRESS OF WHATS TO BE CHANGED
06B2  A5 17               LDA TEMP        ;LOAD ADDR. OF OBJ. THERE
06B4  91 11               STA (INDLOC),Y
06B6  C8                  INY
06B7  A5 18               LDA TEMP+1
06B9  91 11               STA (INDLOC),Y
06BB  A9 FF       DONE    LDA #$FF
06BD  60          OUTE    RTS             ;Z CLEAR IF DONE
06BE                      ;
06BE                      ;
06BE                      ;
06BE  20 00 06    DELETE  JSR SEARCH      ;GET ADDR OF OBJ.
06C1  D0 34               BNE OUTS
06C3  A4 1F               LDY ENTLEN      ;STORE POINTER AT END
06C5  B1 13               LDA (POINTR),Y  ;..OF OBJECT
06C7  85 17               STA TEMP
06C9  C8                  INY
06CA  B1 13               LDA (POINTR),Y
06CC  85 18               STA TEMP+1
06CE  C8                  INY
06CF  A9 00               LDA #0          ;CLEAR OCCUPANCY MARKER
06D1  91 13               STA (POINTR),Y
06D3  A5 10               LDA INDEXD      ;SEE IF REF. TABLE NEEDS
06D5  F0 06               BEQ PREINX      ;..READJUSTING
06D7  20 F6 06            JSR PRETAB
06DA  4C EA 06            JMP MOVEIT
06DD  A5 1B       PREINX  LDA OLD         ;SET FOR CHANGING PREVIOUS
06DF  18                  CLC             ;..ENTRY
```

Figura 9-27 (Continuación).

```
06E0  65 1F          ADC ENTLEN
06E2  85 11          STA INDLOC
06E4  A9 00          LDA #0
06E6  65 1C          ADC OLD+1
06E8  85 12          STA INDLOC+1
06EA  A5 17   MOVEIT LDA TEMP          ;CHANGE WHAT NEEDS CHANGING
06EC  A0 00          LDY #0
06EE  91 11          STA (INDLOC),Y
06F0  C8             INY
06F1  A5 18          LDA TEMP+1
06F3  91 11          STA (INDLOC),Y
06F5  A9 00          LDA #0
06F7  60      OUTS   RTS               ;Z SET IF DONE
06F8          ;
06F8          ;
06F8          ;
06F8  A0 00   PRETAB LDY #0
06FA  B1 15          LDA (OBJECT),Y
06FC  38             SEC               ;REMOVE ASCII LEADER FROM
06FD  E9 41          SBC #$41          ;..FIRST LETTER IN OBJECT
06FF  0A             ASL A             ;MULTIPLY BY 2
0700  18             CLC
0701  65 19          ADC REFBAS        ;INDEX INTO REF. TABLE
0703  85 11          STA INDLOC
0705  A9 00          LDA #0
0707  65 1A          ADC REFBAS+1
0709  85 12          STA INDLOC+1
070B  60             RTS
070C                 .END

ERRORS = 0000 <0000>

SYMBOL TABLE

SYMBOL   VALUE

DELETE   06DE   DONE     06BB   ENTLEN   001F   ENTRY    061D
FOUND    0650   INDEXD   0010   INDLOC   0011   INSERT   06B3
LOOP     0667   LOPE     0685   MORE     0676   MOVEIT   06EA
NEW      0651   NOGOOD   0632   NOTFND   064E   OBJECT   0015
OLD      001B   OUTE     06BD   OUTS     06F7   POINTR   0013
PREINX   06DD   PRETAB   06F8   REFBAS   0019   SEARCH   0600
SETINX   06AF   TABASE   001D   TEMP     0017
END OF ASSEMBLY
```

Figura 9-27 (Continuación).

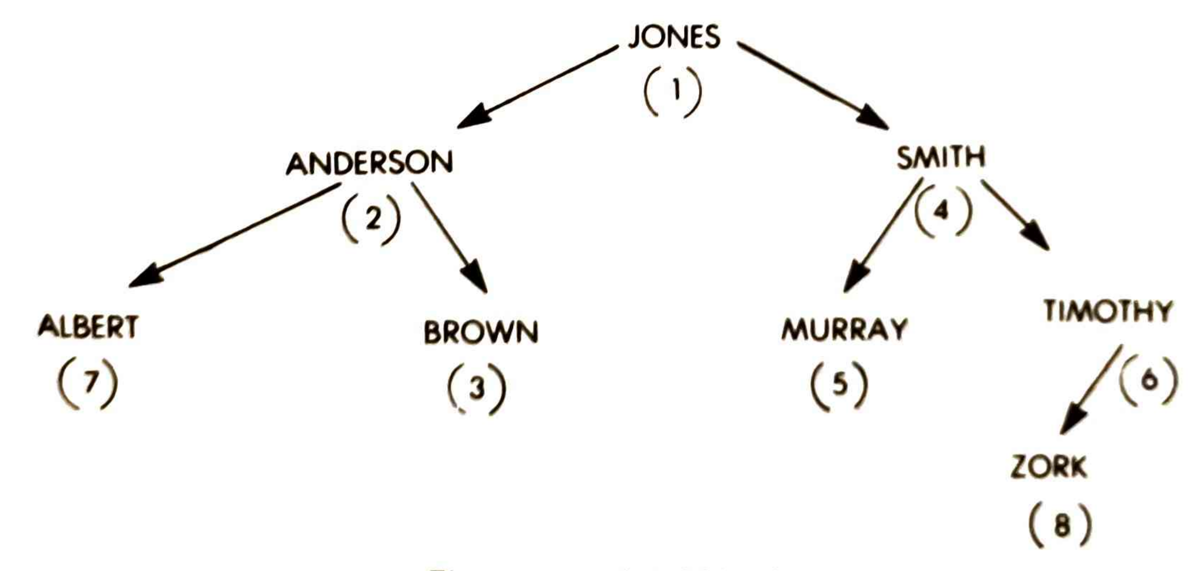


Figura 9-28 Árbol binario.

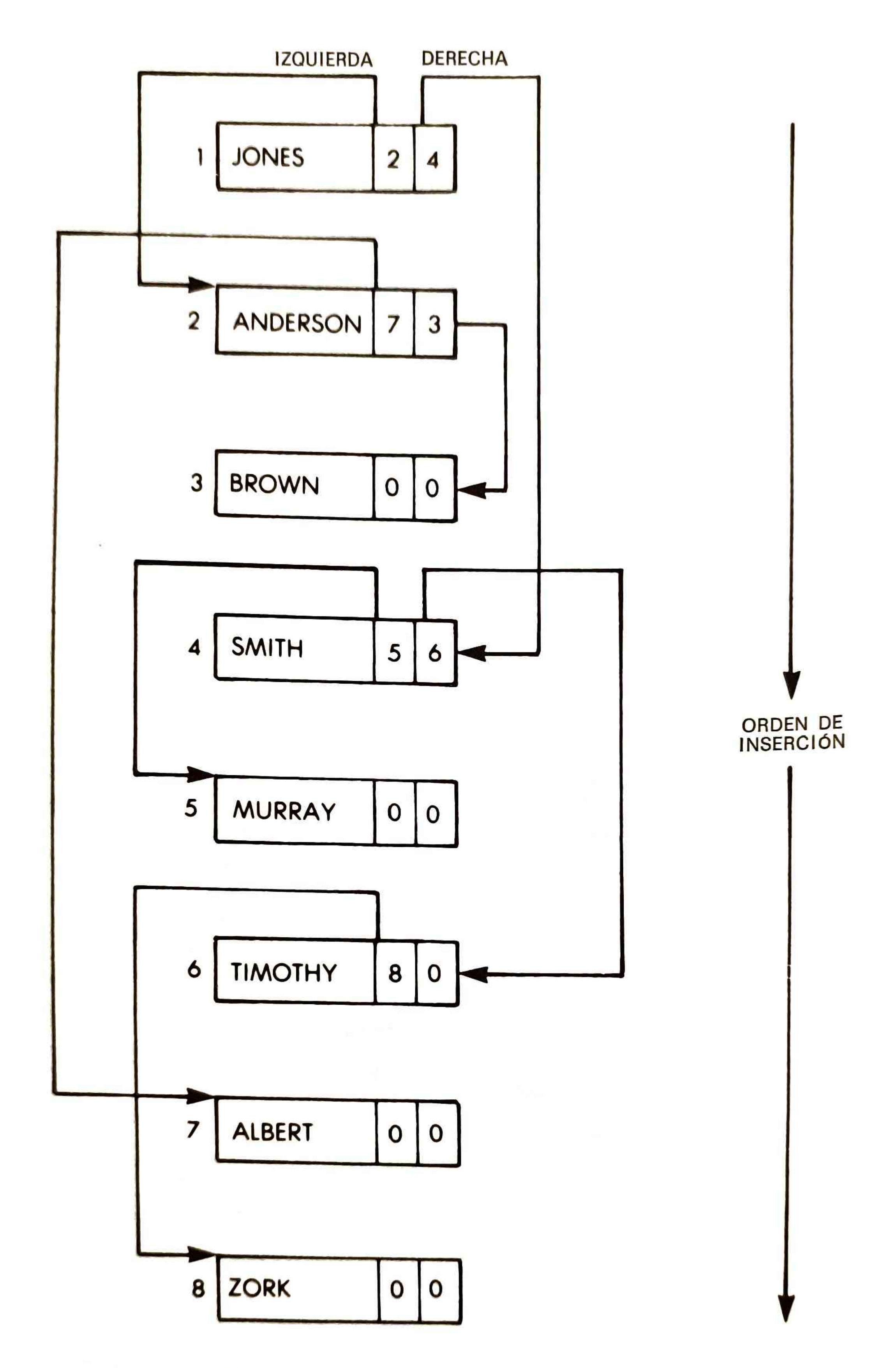


Figura 9-29 Representación en memoria.

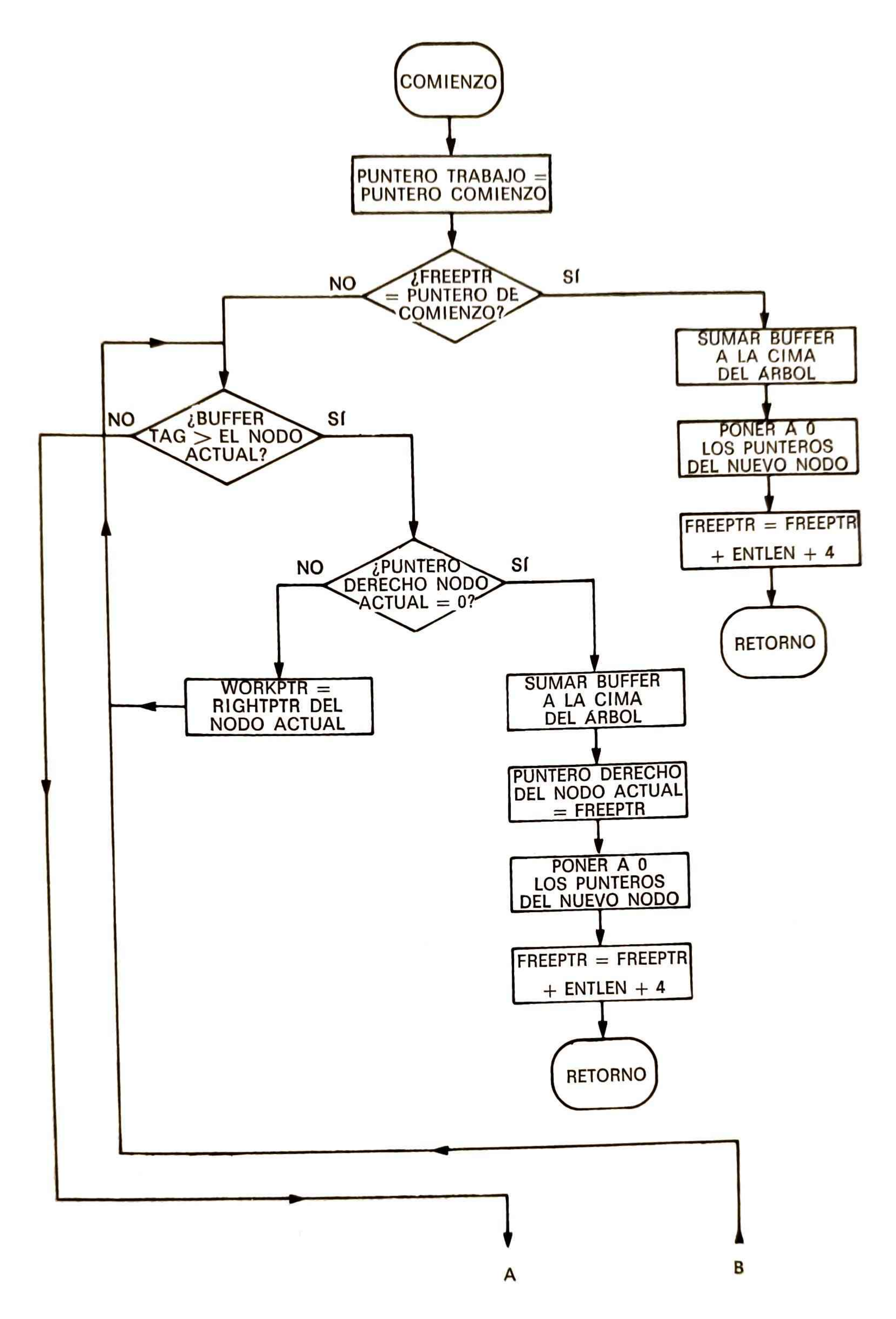


Figura 9-30 Diagrama de flujo de construcción del árbol.

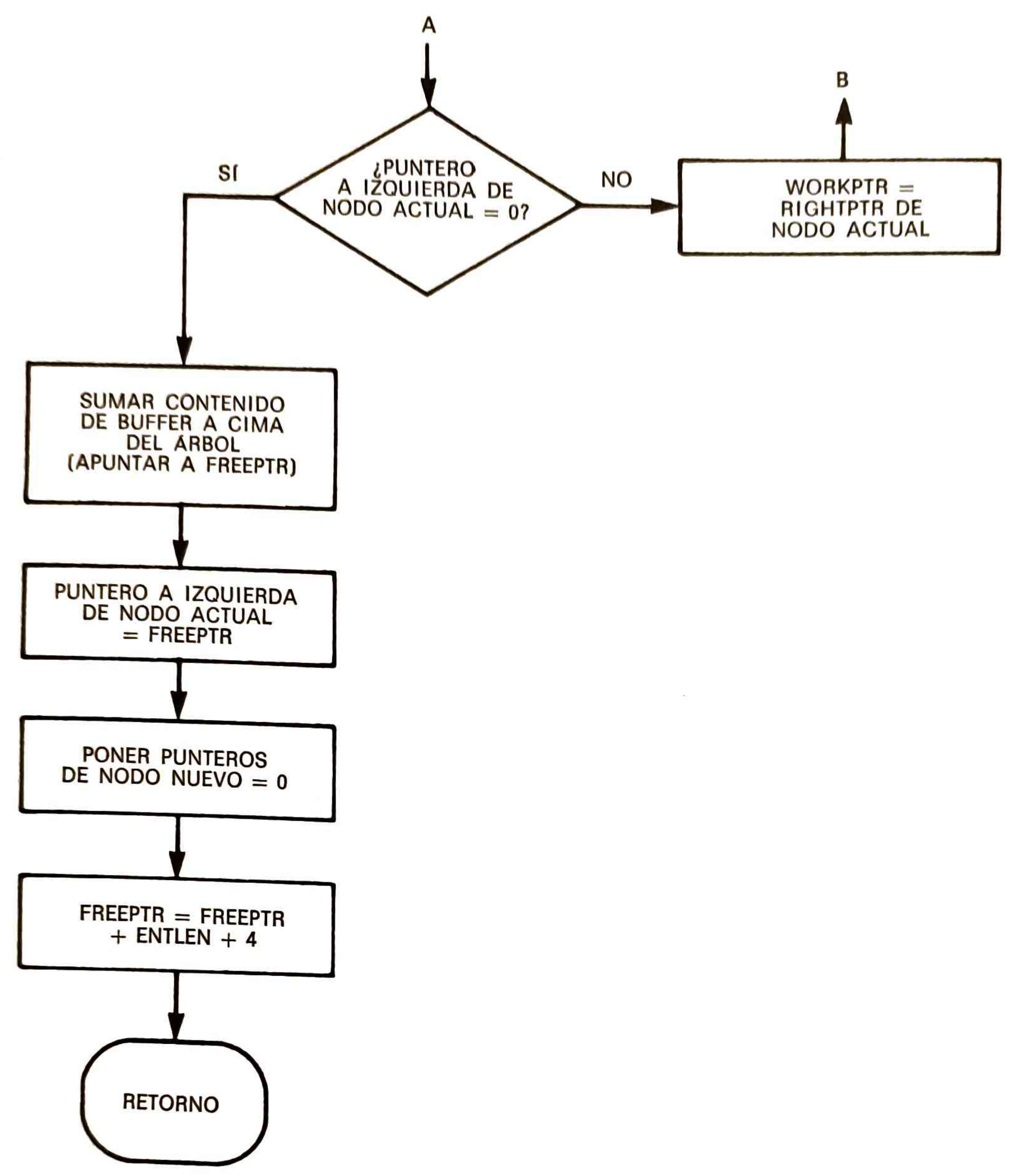


Figura 9-30 (Continuación).

cho y el puntero izquierdo). Para evitar una posible confusión, obsérvese que la representación de la figura 9-29 se ha simplificado y que el puntero derecho aparece en memoria a la izquierda del puntero izquierdo. Las posiciones de memoria utilizadas por este programa se muestran en la figura 9-34 y el programa propiamente dicho en la figura 9-37.

La rutina INSERT reside en las direcciones 0200 a 0282. La etiqueta del objeto a insertar se comparará con el elemento. Si es superior se desplaza a la derecha. Si es inferior, una posición hacia abajo, a la izquierda. El proceso se repetirá hasta que se encuentre un lazo vacío o un "intervalo" ade-

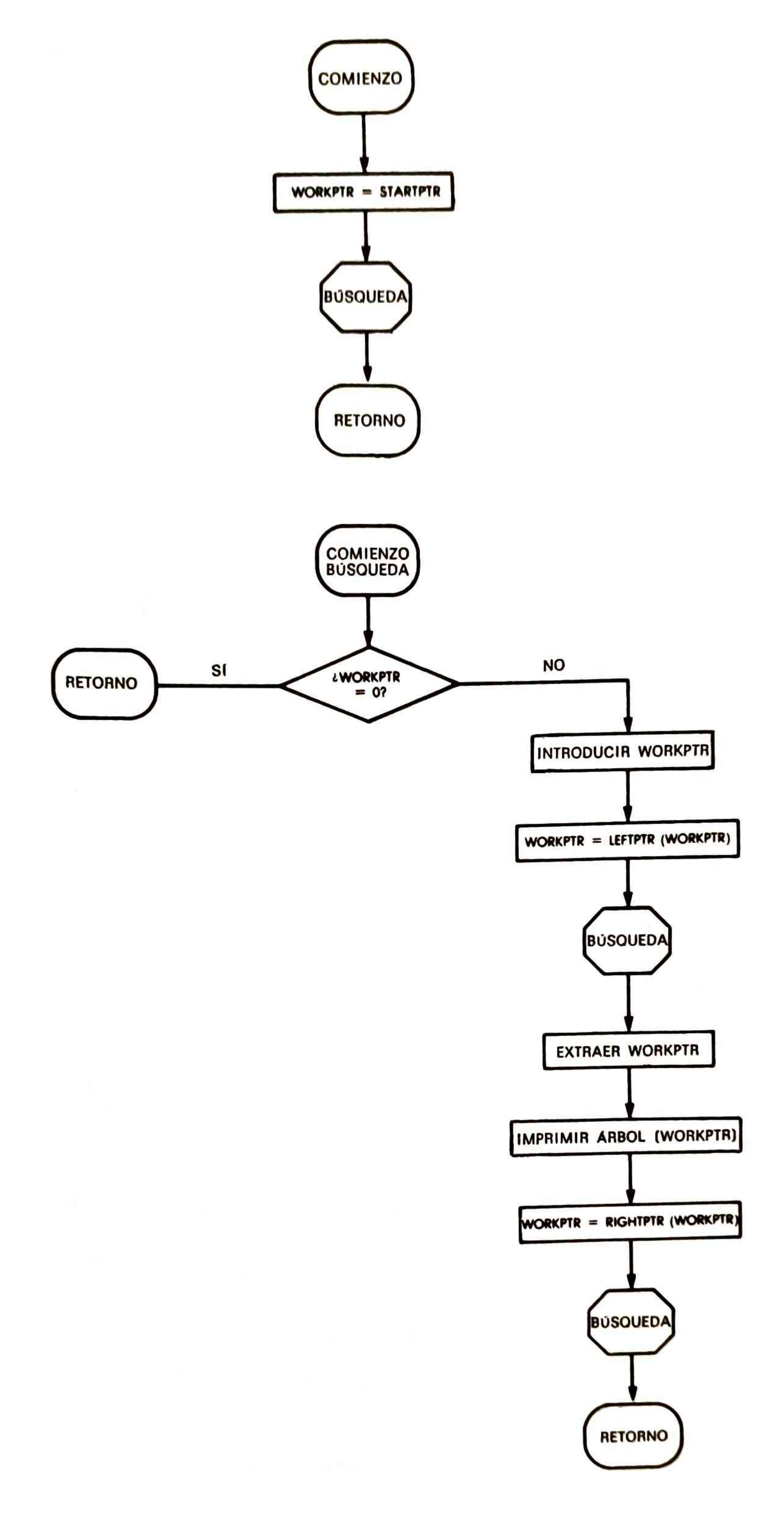


Figura 9-31 Diagrama de flujo de recorrido del árbol.

```
PROGRAMA RECORRIDO DEL ÁRBOL;
COMIENZO
   LLAMADA BÚSQUEDA (PUNTERO DE COMIENZO);
FIN.

BÚSQUEDA DE RUTINA (PUNTERO DE TRABAJO);
COMIENZO
   IF PUNTERO DE TRABAJO = 0 THEN RETURN;
   BÚSQUEDA [LEFTPTR (PUNTERO DE TRABAJO)];
   IMPRIMIR ÁRBOL (PUNTERO DE TRABAJO);
   BÚSQUEDA [RIGHPTR (WORKPTR)];
   RETURN;
FIN.
```

Figura 9-32 Algoritmo de recorrido de un árbol.

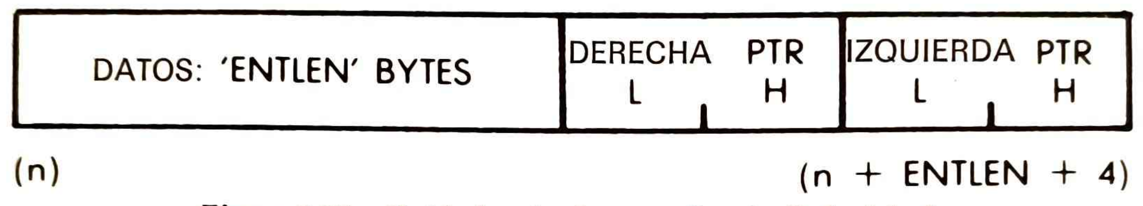


Figura 9-33 Unidades de datos, o "nudos" de árbol.

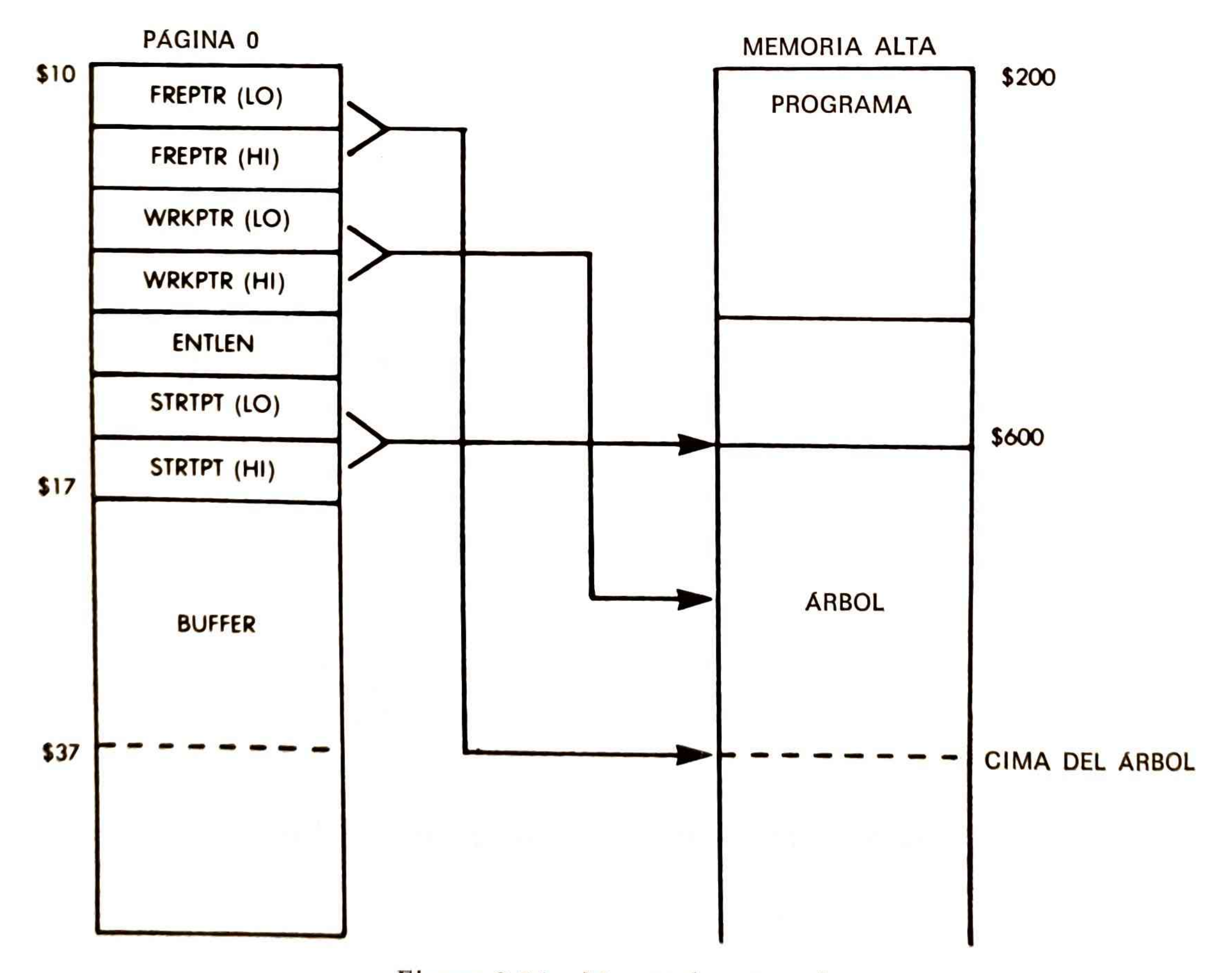


Figura 9-34 Mapas de memoria.

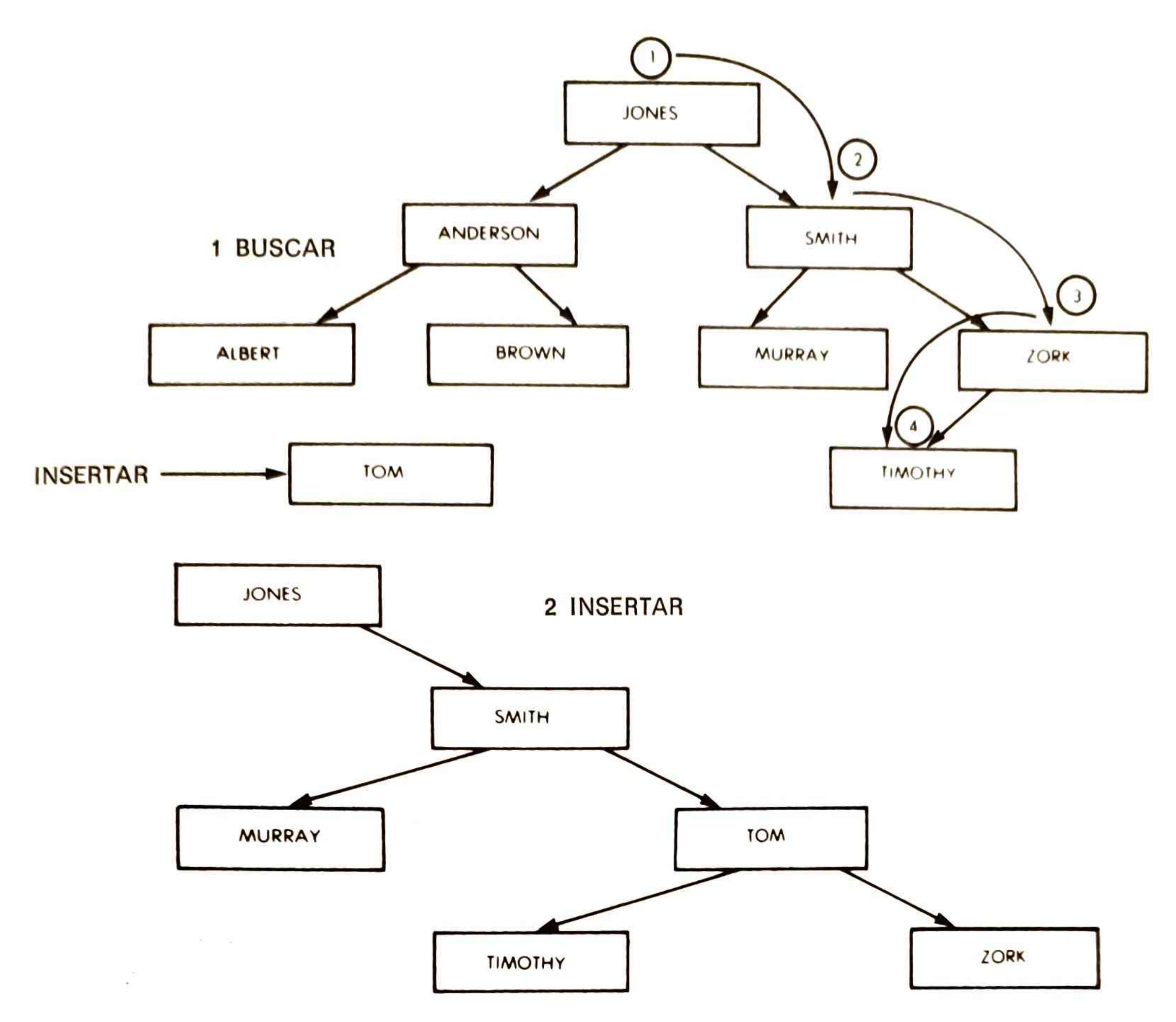


Figura 9-35   Inserción de un elemento en el árbol.

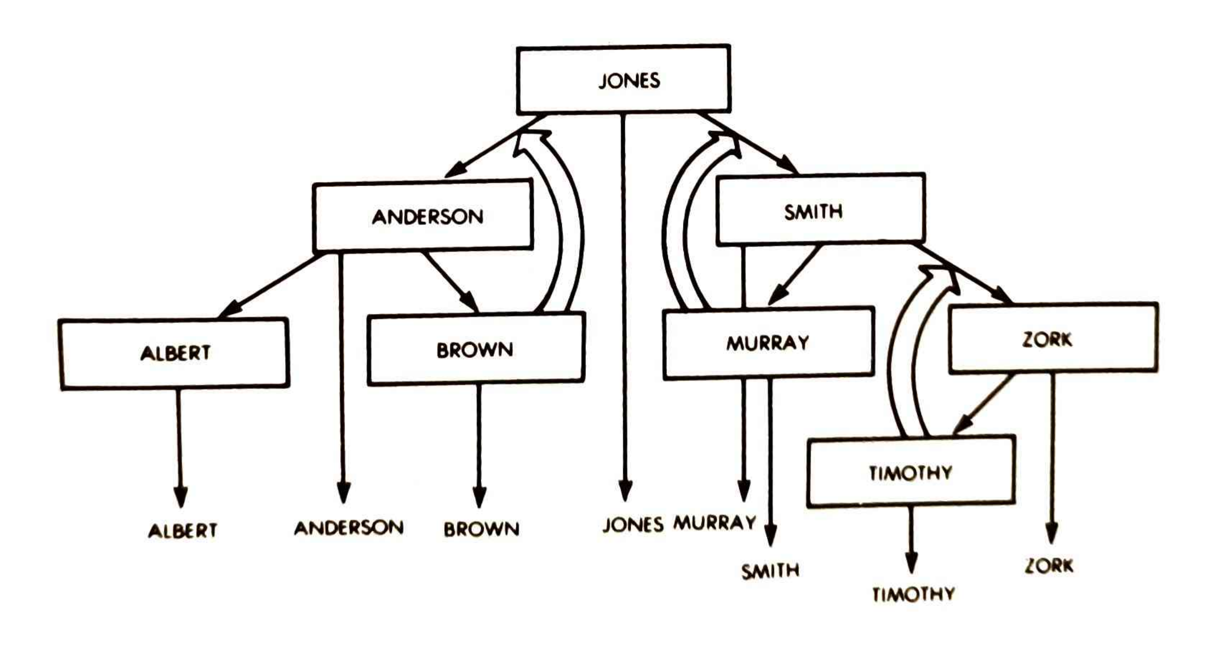


Figura 9-36   Listado del árbol.

cuado para el nudo nuevo (es decir, un nudo es más grande y el otro más pequeño, o viceversa). El nuevo nudo se inserta a continuación, estableciendo simplemente los enlaces adecuados.

La rutina de recorrido (TRAVERSE) reside en las direcciones 0285 a 02D6. Las rutinas de utilidad OUT, ADD y CLRPTR residen en las direcciones 0207 a 02FE (fig. 9-37).

Un ejemplo de inserción en un árbol se muestra en la figura 9-35 y un ejemplo del recorrido de un árbol en la figura 9-36.

```
0000             ;TREE MANAGEMENT PROGRAM.
0000             ;2 ROUTINES: ONE, WHEN CALLED, PLACES
0000             ;THE CONTENTS OF THE BUFFER INTO THE
0000             ;TREE; AND THE SECOND TRAVERSES
0000             ;THE TREE RECURSIVELY, PRINTING ITS
0000             ;NODE CONTENTS IN ALPHANUMERIC ORDER.
0000             ;NOTE: 'ENTLEN' MUST BE INITIALIZED
0000             ;AND 'FREPTR' MUST BE SET EQUAL TO
0000             ;'STRTPTR' BEFORE EITHER ROUTINE IS USED.
0000             ;
0000                    * = $10
0010             FREPTR *=*+2           ;FREE SPACE POINTER: POINTS TO
0012                        ;NEXT FREE LOCATION IN MEMORY.
0012             WRKPTR *=*+2           ;WORKING POINTER, POINTS TO CURRENT NODE.
0014             ENTLEN *=*+1           ;TREE ENTRY LENGTH, IN BYTES.
0015  00 06      STRTPT .WORD $600
0017             BUFFER *=*+20          ;I/O BUFFER.
002B             ;
002B                    * = $200
0200             ;
0200             ;ROUTINE TO BUILD TREE: ADDS ONE DATA UNIT,
0200             ;OR NODE, TO TREE. MUST BE CALLED
0200             ;WITH DATA UNIT TO BE ADDED IN 'BUFFER'.
0200             ;
0200  A5 15      INSERT LDA STRTPT      ;WORKPOINTER <= FREEPOINTER.
0202  85 12             STA WRKPTR
0204  A5 16             LDA STRTPT+1
0206  85 13             STA WRKPTR+1
0208  A5 10             LDA FREPTR      ;IF FREEPOINTER <>
020A  C5 15             CMP STRTPT      ;STARTING LOCATION POINTER,
020C  D0 0D             BNE INLOOP      ;GOTO INSERTION LOOP.
020E  A5 11             LDA FREPTR+1
0210  C5 16             CMP STRTPT+1
0212  D0 07             BNE INLOOP
0214  20 D7 02          JSR ADD         ;LOAD BUFFER INTO CURRENT POSITION.
0217  20 E4 02          JSR CLRPTR      ;SET POINTERS OF CURRENT NODE TO 0.
021A  60                RTS             ;DONE ADDING 1ST NODE.
021B  A0 00      INLOOP LDY #0          ;COMPARE BUFFER TAG TO TAG OF CURRENT
021D  B9 17 00   CMPLP  LDA BUFFER,Y    ;LOCATION...
0220  D1 12             CMP (WRKPTR),Y
0222  90 33             BCC LESSTN      ;BUFR TAG LOWER: ADD BUFFER TO
0224                          ;LEFT SIDE OF TREE.
0224  F0 02             BEQ NXT         ;TAGS EQUAL, TRY NEXT CHR. IN TAGS.
0226  B0 05             BCS GRTNEQ      ;BUFR TAG GREATER, ADD BUFR TO
0228                        ;RIGHT SIDE OF TREE.
0228  C8         NXT    INY
0229  C9 04             CMP #4          ;3 CHRS. COMPARED?
022B  D0 F0             BNE CMPLP       ;NO, CHECK NEXT CHR.
022D  A4 14      GRTNEQ LDY ENTLEN      ;DOES
022F  B1 12             LDA (WRKPTR),Y  ;RIGHT POINTER OF CURRENT NODE = 0 ?
0231  D0 15             BNE NXRNOD      ;IF NOT, MOVE DOWN/RIGHT IN TREE.
0233  C8                INY
```

Figura 9-37 Programas de búsqueda del árbol.

```
0234  B1 12             LDA (WRKPTR),Y
0236  D0 10             BNE NXRNOD
0238  A5 11             LDA FREPTR+1     ;SET RIGHT POINTER OF CURRENT
023A  91 12             STA (WRKPTR),Y   ;NODE = FREEPOINTER.
023C  88                DEY
023D  A5 10             LDA FREPTR
023F  91 12             STA (WRKPTR),Y
0241  20 D7 02          JSR ADD          ;ADD BUFFER TO TREE.
0244  20 E4 02          JSR CLRPTR       ;CLEAR POINTERS OF NEW NODE.
0247  60                RTS              ;DONE, NEW RIGHT NODE ADDED.
0248  A4 14      NXRNOD LDY ENTLEN       ;SET WORKING POINTER
024A  B1 12             LDA (WRKPTR),Y;  RIGHT POINTER OF CURRENT NODE.
024C  AA                TAX
024D  C8                INY
024E  B1 12             LDA (WRKPTR),Y
0250  85 13             STA WRKPTR+1
0252  86 12             STX WRKPTR
0254  4C 1B 02          JMP INLOOP       ;TRY NEW CURRENT NODE.
0257  A4 14      LESSTN LDY ENTLEN       ;DOES LEFT POINTER OF
0259  C8                INY              ;CURRENT NODE = 0 ?
025A  C8                INY
025B  B1 12             LDA (WRKPTR),Y
025D  D0 15             BNE NXLNOD       ;IF SO, MOVE DOWN/LEFT IN TREE.
025F  C8                INY
0260  B1 12             LDA (WRKPTR),Y
0262  D0 10             BNE NXLNOD
0264  A5 11             LDA FREPTR+1     ;SET LEFT POINTER OF CURRENT NODE TO
0266  91 12             STA (WRKPTR),Y   ;POINT TO NEW NODE.
0268  88                DEY
0269  A5 10             LDA FREPTR
026B  91 12             STA (WRKPTR),Y
026D  20 D7 02          JSR ADD          ;ADD NEW NODE CONTENTS.
0270  20 E4 02          JSR CLRPTR       ;CLEAR POINTERS OF NEW NODE.
0273  60                RTS              ;DONE, NEW LEFT NODE ADDED.
0274  A4 14      NXLNOD LDY ENTLEN       ;SET WORKING POINTER =
0276  C8                INY              ;LEFT POINTER OF CURRENT NODE.
0277  C8                INY
0278  B1 12             LDA (WRKPTR),Y
027A  AA                TAX
027B  C8                INY
027C  B1 12             LDA (WRKPTR),Y
027E  85 13             STA WRKPTR+1
0280  86 12             STX WRKPTR
0282  4C 1B 02          JMP INLOOP       ;TRY NEW CURRENT NODE.
0285                    ;
0285                    ;TREE TRAVERSER : LISTS NODES OF TREE
0285                    ;IN ALPHANUMERICAL ORDER.
0285                    ;OUTPUT ROUTINE TO XFER BUFFER TO OUTPUT
0285                    ;DEVICE IS NEEDED.
0285                    ;
0285  A5 15      TRVRSE LDA STRTPT       ;WORKING POINTER <= START POINTER.
0287  85 12             STA WRKPTR
0289  A5 16             LDA STRTPT+1
028B  85 13             STA WRKPTR+1
028D  A5 13      SEARCH LDA WRKPTR+1
028F  A6 12             LDX WRKPTR       ;IF WORKING POINTER <> 0,
0291  D0 07             BNE OK           ;CONTINUE;
0293  A4 13             LDY WRKPTR+1
0295  D0 03             BNE OK
0297  4C C6 02          JMP RETN         ;ELSE, RETURN.
029A  48         OK     PHA              ;PUSH WORKING POINTER
029B  8A                TXA              ;ONTO STACK.
029C  48                PHA
029D  A4 14             LDY ENTLEN       ;SET WORKING POINTER =
029F  C8                INY              ;LEFT POINTER OF CURRENT NODE.
02A0  C8                INY
02A1  B1 12             LDA (WRKPTR),Y
02A3  AA                TAX
02A4  C8                INY
02A5  B1 12             LDA (WRKPTR),Y
```

Figura 9-37 (Continuación).

```
02A7  85 13            STA WRKPTR+1
02A9  86 12            STX WRKPTR
02AB  20 8D 02         JSR SEARCH      ;SEARCH NEW NODE, RECURSIVELY.
02AE  68               PLA             ;POP OLD CURRENT NODE INTO WORKING POINTER.
02AF  85 12            STA WRKPTR
02B1  68               PLA
02B2  85 13            STA WRKPTR+1
02B4  20 C7 02         JSR OUT         ;OUTPUT CURRENT NODE CONTENTS.
02B7  A4 14            LDY ENTLEN      ;SET WORKING POINTER =
02B9  B1 12            LDA (WRKPTR),Y  ;CURRENT NODE'S RIGHT POINTER.
02BB  AA               TAX
02BC  C8               INY
02BD  B1 12            LDA (WRKPTR),Y
02BF  85 13            STA WRKPTR+1
02C1  86 12            STX WRKPTR
02C3  20 8D 02         JSR SEARCH      ;SEARCH NEW NODE.
02C6  60        RETN   RTS             ;DONE, RETURN.
02C7            ;
02C7            ;BUFFER OUTPUT ROUTINE.
02C7            ;
02C7  A0 00     OUT    LDY #0
02C9  B1 12     XFR    LDA (WRKPTR),Y  ;GET CHR. FROM CURRENT NODE.
02CB  99 17 00         STA BUFFER,Y    ;PUT IN BUFFER.
02CE  C8               INY             ;REPEAT UNTIL...
02CF  C4 14            CPY ENTLEN      ;ALL CHARACTERS XFERRED.
02D1  D0 F6            BNE XFR
02D3  EA               NOP             ;INSERT CALL TO SUBROUTINE
02D4  EA               NOP             ;WHICH OUTPUTS BUFFER HERE.
02D5  EA               NOP
02D6  60               RTS             ;DONE.
02D7            ;
02D7            ;ROUTINE WHICH PLACES BUFFER
02D7            ;CONTENTS IN NEW NODE.
02D7            ;
02D7  A0 00     ADD    LDY #0
02D9  B9 17 00  MOV    LDA BUFFER,Y    ;GET CHR. FROM BUFFER.
02DC  91 10            STA (FREPTR),Y  ;STORE IN NEW NODE.
02DE  C8               INY             ;REPEAT UNTIL...
02DF  C4 14            CPY ENTLEN      ;ALL CHRS XFERRED.
02E1  D0 F6            BNE MOV
02E3  60               RTS             ;DONE.
02E4            ;
02E4            ;ROUTINE TO CLEAR POINTERS OF NEW NODE,
02E4            ;AND UPDATE FREE SPACE POINTER.
02E4            ;
02E4  A4 14     CLRPTR LDY ENTLEN      ;SET UP INDEX TO POINT
02E6                             ;TO TOP OF POINTER LOCATIONS.
02E6  A9 00            LDA #0
02E8  A2 04            LDX #4          ;LOOP 4X TO CLEAR POINTERS
02EA  91 10     CLRLP  STA (FREPTR),Y  ;CLEAR POINTER LOCATION.
02EC  C8               INY             ;POINT TO NEXT POINTER LOCATION.
02ED  CA               DEX
02EE  D0 FA            BNE CLRLP       ;LOOP IF NOT DONE.
02F0  A5 14            LDA ENTLEN      ;GET ENTRY LENGTH,
02F2  18               CLC             ;AND ADD 4 FOR POINTER SPACE.
02F3  69 04            ADC #4
02F5  65 10            ADC FREPTR      ;ADD TO FREE SPACE POINTER TO
02F7  90 02            BCC CC          ;UPDATE IT.
02F9  E6 11            INC FREPTR+1    ;TAKE CARE OF OVERFLOWS.
02FB  85 10     CC     STA FREPTR      ;RESTORE UPDATED FREE SPACE PTR.
02FD  60               RTS             ;DONE.
02FE                   .END

ERRORS = 0000 <0000>
END OF ASSEMBLY
```

Figura 9-37 (Continuación).

Notas sobre los árboles

Los árboles binarios se pueden construir y recorrer de muchas formas. Por ejemplo, otra representación para nuestro árbol podría ser:

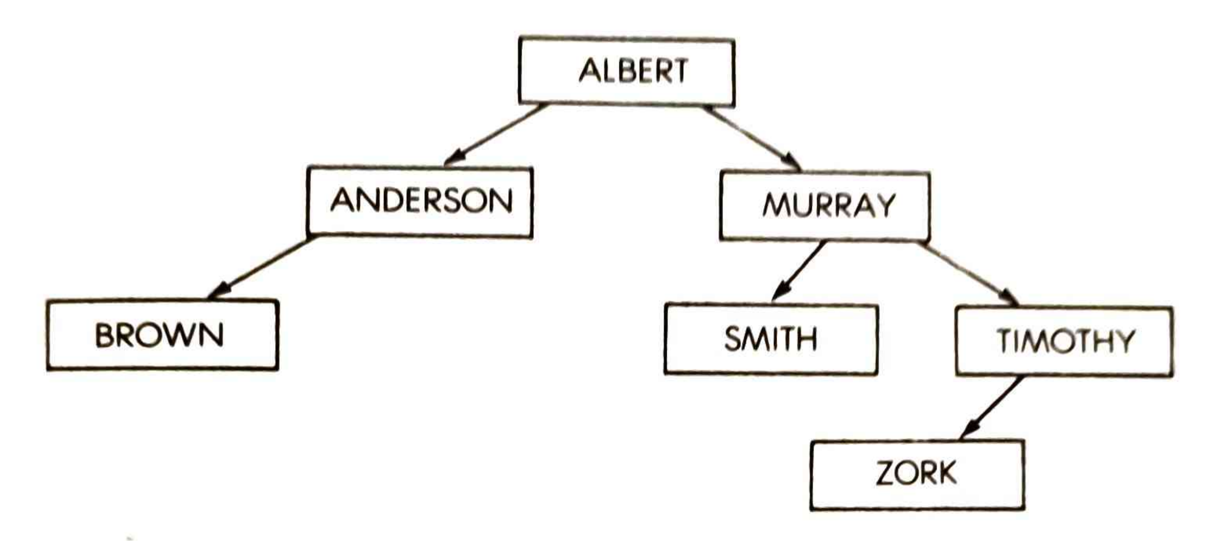


Figura 9-38 Árbol previamente ordenado.

Entonces, sería preciso recorrerlo en "orden":

1 — listar la raíz
2 — recorrer el subárbol de la izquierda
3 — recorrer el subárbol de la derecha.

Existen otras muchas técnicas y convenios.

ALGORITMO DE CLASIFICACIÓN ALEATORIA

Un problema común cuando se crean estructuras de datos es cómo situar identificadores de modo sistemático en un espacio limitado de memoria, de forma que se pueda acceder a ellos fácilmente. Lamentablemente, a menos que los identificadores sean números secuenciales consecutivos, no se les puede situar bien en memoria sin lagunas. En particular, si los nombres se situaron en memoria de modo que se puede acceder a ellos más fácilmente (es decir, se situaron por orden alfabético), eso exigirá un tamaño enorme de memoria; se tendría que reservar un solo bloque de memoria para cada nombre posible. Esto no es aceptable. Para resolver este problema se puede utilizar un algoritmo de clasificación aleatoria para asignar un número único

(o casi) a todo nombre que se haya introducido en memoria. La función matemática utilizada para realizar el algoritmo de clasificación aleatoria debe ser sencilla de modo que el algoritmo pueda ser rápido, pero lo bastante perfeccionado para que sea aletoria la distribución de los nombres posibles sobre el espacio de memoria disponible. El número resultante se puede utilizar como un índice de la posición efectiva y su acceso será posible rápidamente. Y por esta razón, este algoritmo se utiliza corrientemente para las pseudoinstrucciones de nombres alfabéticos.

Ya que ningún algoritmo puede garantizar que dos nombres no se clasifiquen aleatoriamente en la misma posición de memoria (una "colisión") se debe desarrollar una técnica para resolver el problema de las colisiones. Un buen algoritmo de clasificación aleatoria repartirá los nombres uniformemente sobre el espacio de memoria disponible, y permitirá un acceso eficaz de sus valores una vez que hayan sido almacenados en una tabla. El algoritmo de clasificación aleatoria utilizado en este caso es muy sencillo y basta realizar la operación OR exclusiva de todos los bytes del indicativo de clasificación. Después de cada suma se realiza una rotación para mejorar la aleatoriedad.

La técnica utilizada para resolver el problema de las colisiones es secuencial simple. Se llama "técnica de direccionamiento abierto secuencial"; el siguiente bloque disponible secuencialmente en la tabla se asigna a la entrada. Ello se puede comparar a una agenda de direcciones. Supongamos que se debe introducir una nueva entrada SMITH, pero que la página "S" está completa en nuestra agenda de direcciones. Se utilizará la siguiente

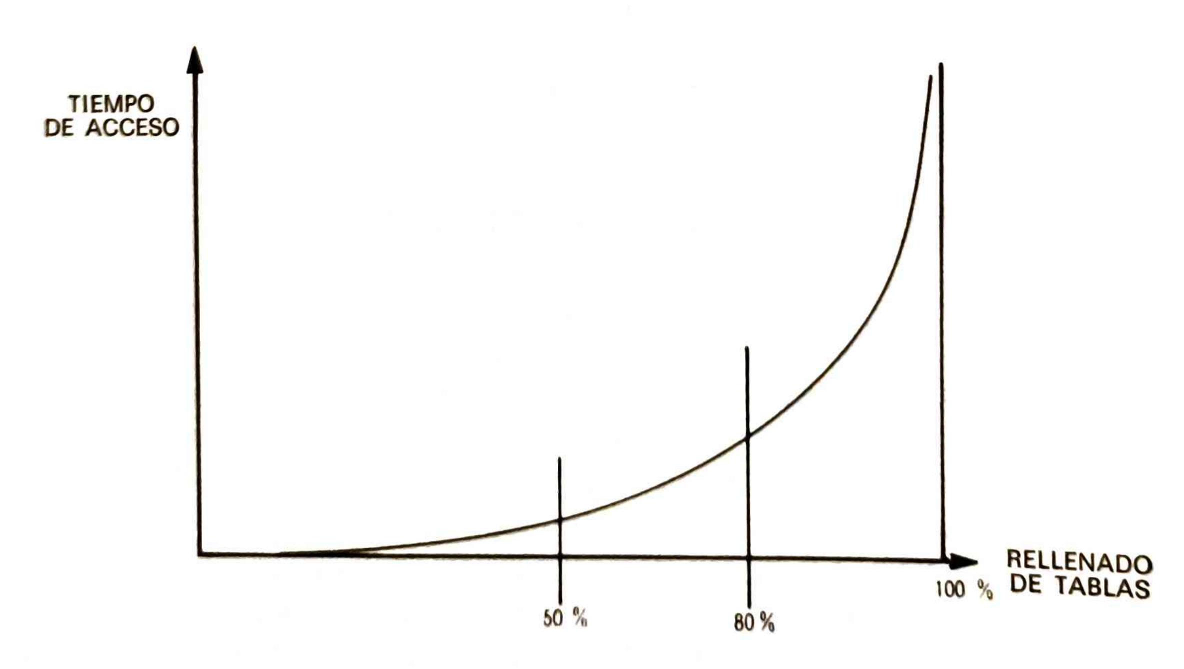


Figura 9-39 Tiempo de acceso en función del relleno relativo.

página secuencial (en este caso "T"). Obsérvese que no es inevitable otra colisión con una nueva entrada que comience con una "T"; la entrada que comienza por "S" se puede eliminar antes de que se necesite introducir cualquier entrada que comience por "T".

Se puede observar también que habrá una cadena de colisiones. Si la cadena es larga, y la tabla no está completa, el algoritmo de clasificación aleatoria es un mal diseño.

Ya que es conveniente utilizar una potencia de dos para el formato de datos, la longitud del formato es de ocho caracteres; seis son para el indicativo y dos para los datos. Esta es una situación corriente cuando se crea la tabla de símbolos de un ensamblador. Al símbolo se asignan hasta seis símbolos hexadecimales y dos a la dirección que representa (2 bytes).

Cuando se accede a elementos desde la tabla de información aleatoriamente clasificada, el tiempo requerido para la exploración no depende del tamaño de la tabla sino del grado en que se ha rellenado la tabla. Generalmente, conservando la tabla a menos del 80 % de su capacidad total se asegurará un tiempo de acceso alto (uno o dos ensayos). Es responsabilidad de la rutina de llamada conservar el registro del grado de ocupación de la tabla y evitar todo desbordamiento.

En la figura 9-39 se muestra el aumento del tiempo de acceso en función

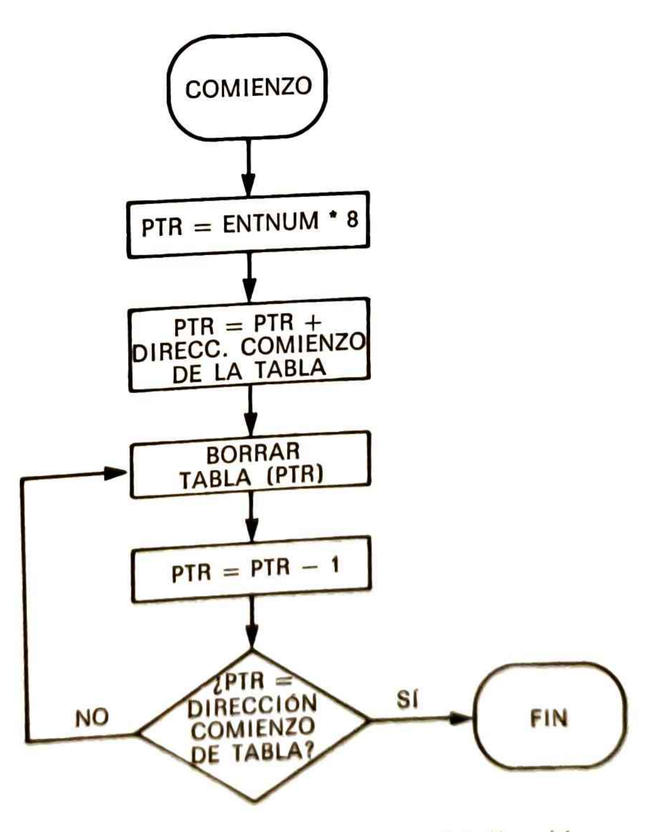


Figura 9-40 Subrutina de inicialización.

del rellenado de la tabla. Las rutinas principales utilizadas por el programa son la subrutina de inicialización (INIT), (fig. 9-40), la rutina de almacenamiento (fig. 9-41), la rutina de acceso (fig. 9-42), y la rutina de algoritmo de clasificación aleatoria (fig. 9-43). La posición de memoria se muestra en la figura 9-44 y el programa se da en la figura 9-45. El programa es destinado a mostrar todos los principales algoritmos utilizados en el mecanismo de algoritmo de clasificación aleatoria. Si estos programas se han de incorporar a una realización real, se recomienda encarecidamente añadir las funciones habituales auxiliares de gestión para evitar situaciones imprevistas.

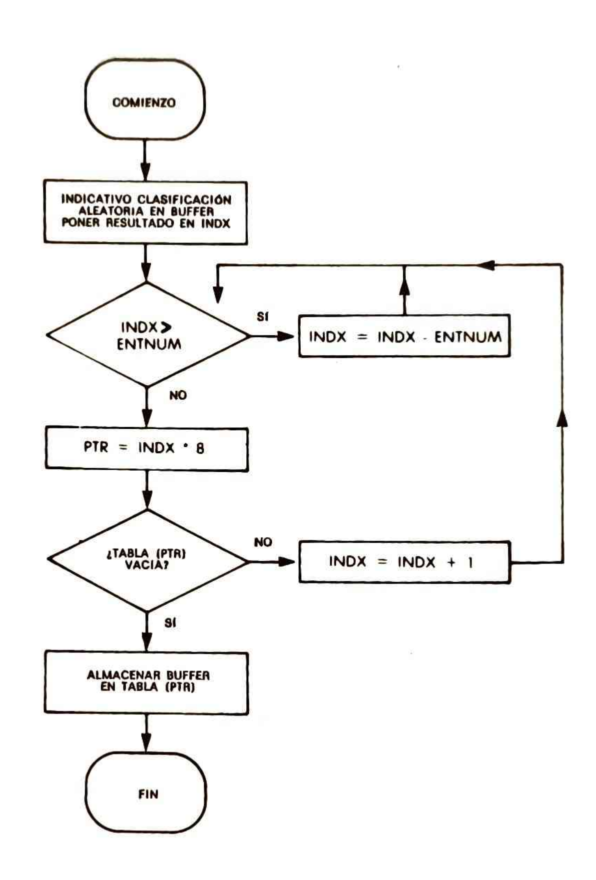


Figura 9-41 Rutina de almacenamiento (store).

En particular, se debe impedir la posibilidad de una tabla completa o de un indicativo de clasificación incorrecto, ya que ello puede originar lazos infinitos en el programa. El lector ha de estar dispuesto para estudiar este programa. No solamente porque desmitifica un algoritmo de clasificación aleatoria sino también porque resuelve un problema práctico importante que aparece cuando se diseña un ensamblador o cualquier otra estructura, en donde las tablas de nombres con sus valores equivalentes se deben guardar de un modo eficaz.

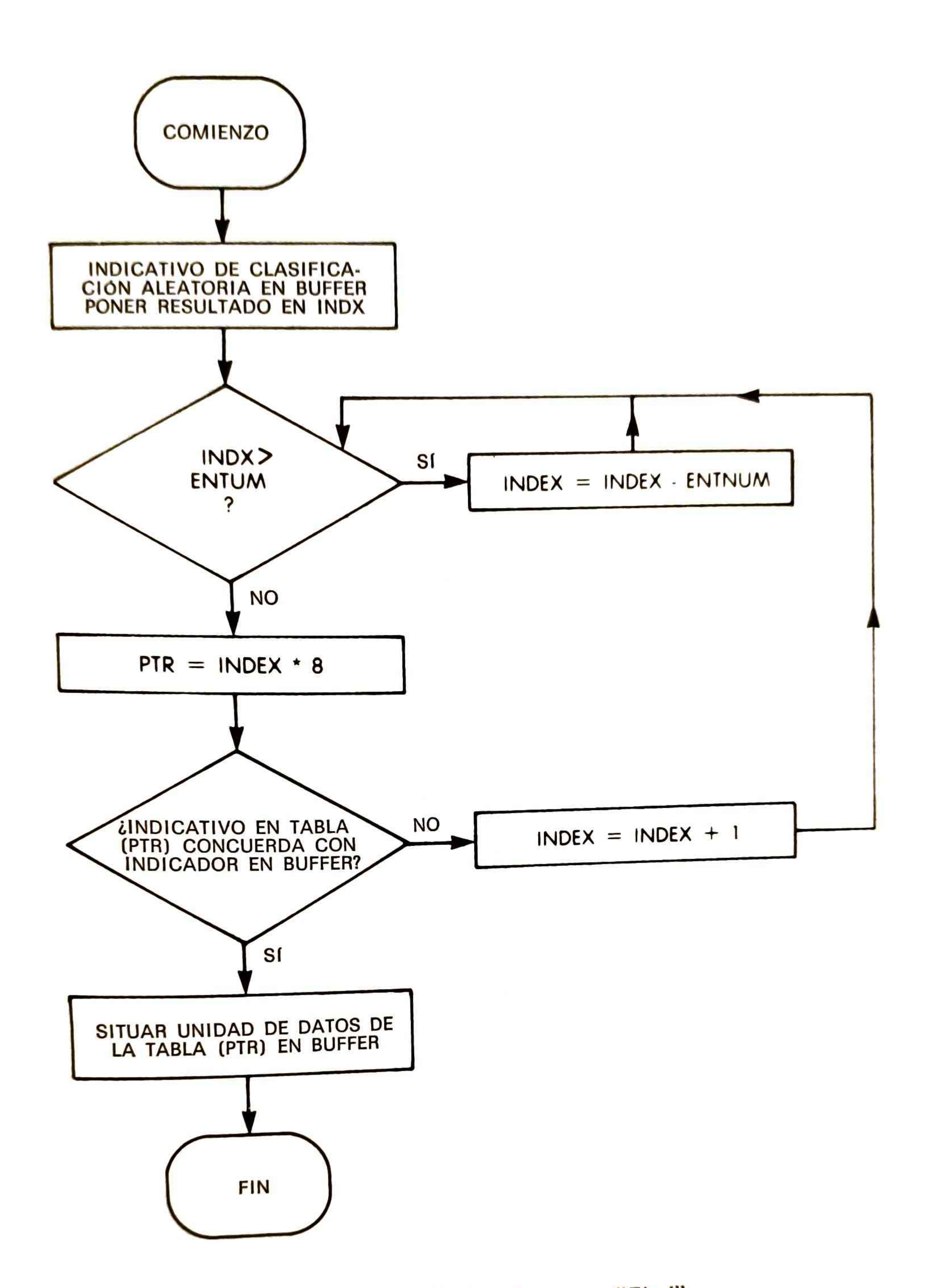


Figura 9-42 Rutina de acceso "Find".

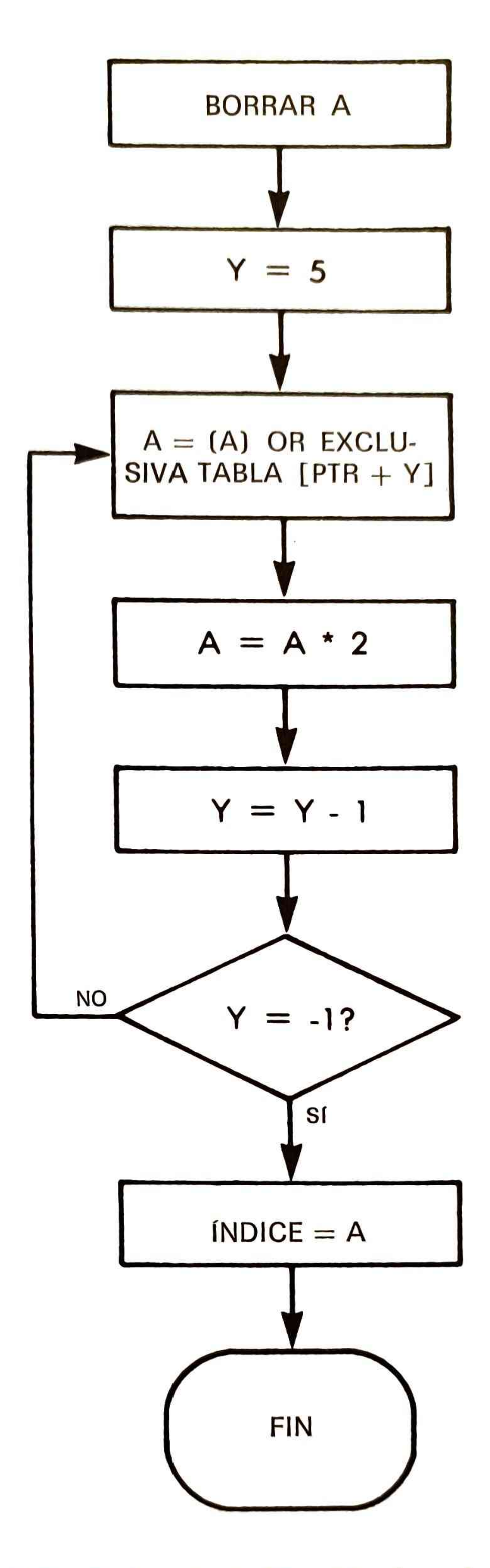


Figura 9-43 Rutina de clasificación aleatoria (hash).

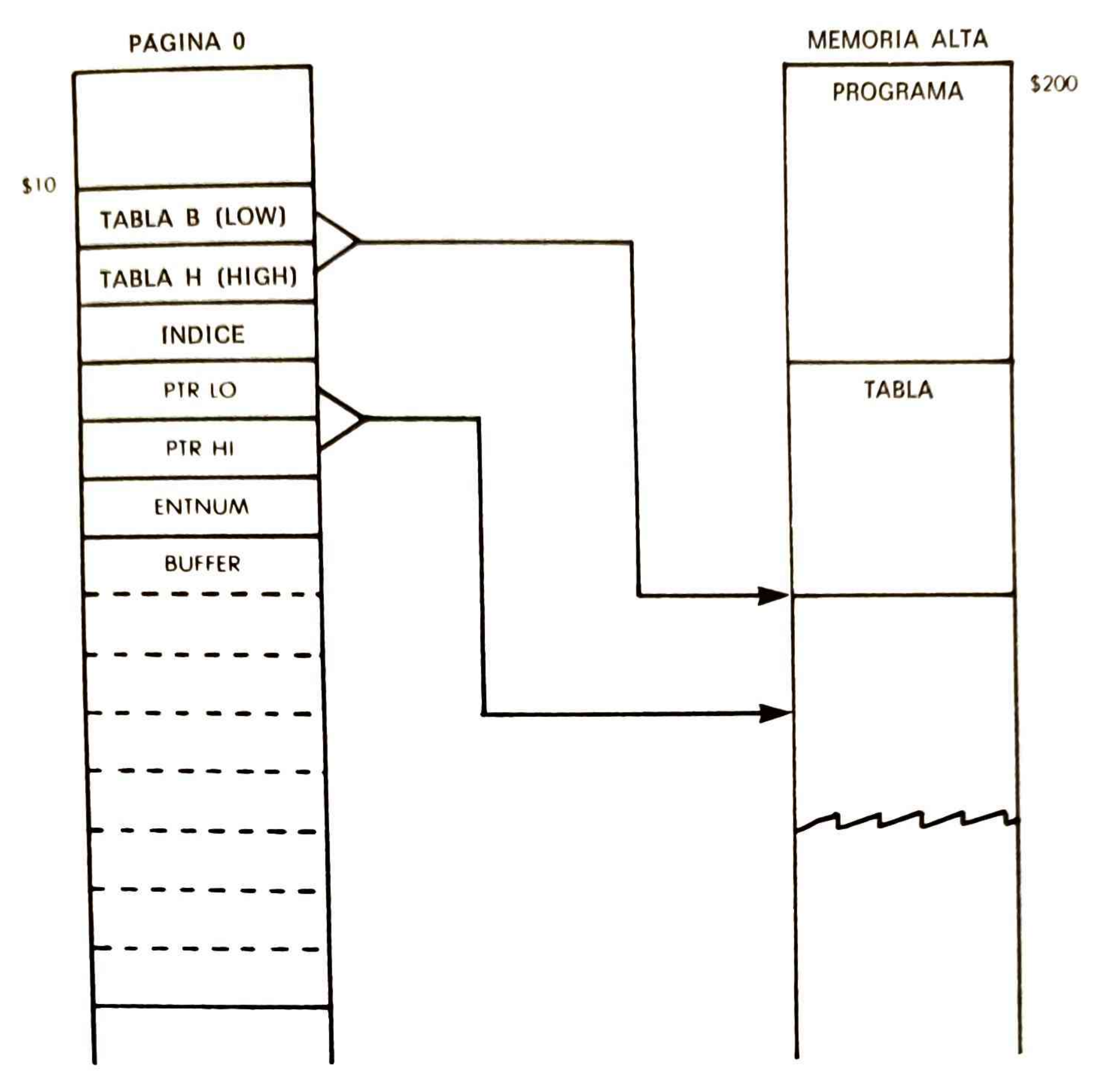


Figura 9-44 Almacenamiento/acceso de clasificación aleatoria: mapas de memoria.

CLASIFICACIÓN DE BURBUJA

La clasificación de burbuja es una técnica de clasificación utilizada para disponer los elementos de una tabla en orden ascendente o descendente. La técnica de clasificación de burbuja deriva su nombre del hecho que el elemento más pequeño sube a la cima de la tabla como si se tratara de una burbuja de gas. Cada vez que "choca" con un elemento "más pesado" salta por encima del mismo.

Un ejemplo práctico de clasificación de burbuja se muestra en la figura 9-46. La lista a clasificar contiene: 10, 5, 0, 2 y 100, y se debe clasificar en orden descendente ("0" en la parte superior). El algoritmo es sencillo y el diagrama de flujo se muestra en la figura 9-47.

```
LINE # LOC     CODE       LINE

       0000              ;PROGRAM TO STORE ASSEMBLER SYMBOLS IN A
       0000              ;TABLE, ACCESSED BY HASHING. THE SYMBOLS
       0000              ;ARE 6 CHRS, DATA 2. THE MAXIMUM NUMBER OF
       0000              ;8-BYTE UNITS TO BE STORED IN THE TABLE
       0000              ;SHOULD BE IN 'ENTNUM', BEGINNING ADDRESS OF
       0000              ;TABLE SHOULD BE IN 'TABLE'. NOTE THAT
       0000              ;TABLE MUST BE INITIALIZED WITH ROUTINE
       0000              ;'INIT' PRIOR TO USE.
       0000              ;IT IS THE RESPONSIBILITY OF THE CALLING
       0000              ;PROGRAM NO TO EXCEED THE TABLE SIZE.
       0000              ;
       0000                     * = $10
       0010 00 06        TABLE  .WORD $600     ;STARTING ADDRESS OF TABLE.
       0012              INDX   *=*+1          ;NUMBER OF DATA UNIT TO BE ACCESSED.
       0013              PTR    *=*+2          ;POINTER TO DATA UNIT IN TABLE.
       0015              ENTNUM *=*+1          ;NUMBER OF ENTRIES IN TABLE (256 MAX)
       0016              BUFFER *=*+8          ;INPUT/ OUTPUT BUFFER.
       001E              ;
       001E                     * = $200
       0200              ;
       0200              ;ROUTINE 'INIT' : INITIALIZES TABLE
       0200              ;TO ZEROES.
       0200              ;
       0200 A5 15        INIT   LDA ENTNUM
       0202 85 13               STA PTR        ;STORE # OF ENTRIES IN POINTER
       0204 20 72 02            JSR SHADD      ;MULTIPLY PTR*8, ADD TABLE POINTER.
       0207 A2 00               LDX #0         ;CLEAR X FOR INDIRECT ADDRESSING.
       0209 A9 00        CLRLP  LDA #0         ;GET CLEARING CONSTANT
       020B A4 13               LDY PTR
       020D D0 02               BNE DECR       ;IF PTR <> 0, DON'T DECREMENT HI BYTE.
       020F C6 14               DEC PTR+1      ;DECREMENT HI BYTE OF POINTER.
       0211 C6 13        DECR   DEC PTR        ;DECREMENT LO BYTE.
       0213 81 13               STA (PTR,X)    ;CLEAR LOCATION.
       0215 A5 13               LDA PTR        ;CHECK IF POINTER = TABLE POINTER,
       0217 C5 10               CMP TABLE      ;IF UNEQUAL, CLEAR NEXT LOCATION.
       0219 D0 EE               BNE CLRLP
       021B A5 14               LDA PTR+1
       021D C5 11               CMP TABLE+1
       021F D0 E8               BNE CLRLP
       0221 60                  RTS
       0222              ;
       0222              ;ROUTINE 'STORE': PLACES BUFFER CONTENTS IN
       0222              ;TABLE, USING 1ST 6 CHRS. OF BUFFER AS A
       0222              ;'KEY' TO DETERMINE HASHED ADDRESS IN
       0222              ;TABLE.
       0222              ;
       0222 A2 00        STORE  LDX #0         ;CLEAR X FOR INDEXED ADDRESSING.
       0224 20 90 02            JSR HASH       ;GET HASHED INDEX..
       0227 20 62 02     CMPR1  JSR LIMIT      ;MAKE SURE INDEX IS WITHIN BOUNDS.
       022A A1 13               LDA (PTR,X)    ;CHECK DATA UNIT...
       022C F0 05               BEQ EMPTY      ;JUMP IF EMPTY.
       022E E6 12               INC INDX       ;TRY NEXT UNIT.
       0230 4C 27 02            JMP CMPR1      ;CHECK FOR NEXT UNIT INDEX VALID.
       0233 A0 07        EMPTY  LDY #7         ;LOOP 8X TO LOAD DATA UNIT.
       0235 B9 16 00     FILL   LDA BUFFER,Y   ;GET CHR FROM BUFFER,
       0238 91 13               STA (PTR),Y    ;PLACE IT IN BUFFER.
       023A 88                  DEY
       023B 10 F8               BPL FILL       ;XFER NEXT CHR.
       023D 60                  RTS            ;ADDITION DONE.
       023E              ;
       023E              ;ROUTINE 'FIND' :
       023E              ;FINDS ENTRY WHOSE KEY IS IN BUFFER.
       023E              ;ENTRY, WHEN FOUND, IS COPIED INTO
       023E              ;BUFFER, ALONG WITH 2 BYTES OF DATA.
       023E              ;
       023E A2 00        FIND   LDX #0         ;CLEAR X FOR INDIRECT ADDRESSING.
       0240 20 90 02            JSR HASH       ;GET HASH PRODUCT.
       0243 20 62 02     CMPR2  JSR LIMIT      ;MAKE SURE RESULT IS WITHIN LIMITS
```

Figura 9-45 Programa de clasificación aleatoria.

```
0246  A0 05            LDY #5        ;LOOP 6X TO COMPARE BUFFER TO DATA ITEM.
0248  B1 13      CHKLP LDA (PTR),Y   ;GET CHR FROM TABLE.
024A  D9 16 00         CMP BUFFER,Y  ;IS IT = BUFFER CHR?
024D  D0 0E            BNE BAD       ;IF NOT, TRY NEXT DATA UNIT.
024F  88               DEY
0250  10 F6            BPL CHKLP     ;CHECK NEXT CHRS.
0252  A0 07      MATCH LDY #7        ;LOOP 8X TO XFER CHRS TO BUFFER.
0254  B1 13      XFER  LDA (PTR),Y   ;GET CHR. FROM TABLE.
0256  99 16 00         STA BUFFER,Y  ;STORE IN BUFFER.
0259  88               DEY
025A  10 F8            BPL XFER      ;LOOP TO XFER CHRS.
025C  60               RTS           ;DONE :DATA UNIT FOUND, IN BUFFER.
025D  E6 12      BAD   INC INDX      ;NOT FOUND, TRY NEXT DATA UNIT.
025F  4C 43 02         JMP CMPR2     ;VALIDATE NEW DATA UNIT INDEX.
0262             ;
0262             ;ROUTINE TO MAKE SURE DATA INDEX IS WITHIN
0262             ;BOUNDS SET BY ENTNUM, THEN MULTIPLY INDEX
0262             ;BY 8, AND ADD IT TO TABLE POINTER. THE
0262             ;RESULT IS PLACED IN 'PTR' AS DATA UNIT ADDRESS.
0262             ;
0262  A5 12      LIMIT LDA INDX      ;GET INDEX.
0264  C5 15      TEST  CMP ENTNUM    ;INDEX > NUMBER OF DATA ITEMS?
0266  90 06            BCC OK        ;JUMP IF NOT.
0268  38               SEC           ;YES -
0269  E5 15            SBC ENTNUM    ;SUBTRACT # OF ITEMS UNTIL
026B  4C 64 02         JMP TEST      ;INDEX WITHIN BOUNDS.
026E  85 13      OK    STA PTR       ;STORE GOOD INDEX IN POINTER.
0270  85 12            STA INDX      ;SAVE UPDATED INDEX.
0272  A9 00      SHADD LDA #0        ;CLEAR UPPER POINTER FOR SHIFT.
0274  85 14            STA PTR+1
0276  06 13            ASL PTR       ;SHIFT PTR 3X LEFT - MULTIPLY BY 8.
0278  26 14            ROL PTR+1
027A  06 13            ASL PTR
027C  26 14            ROL PTR+1
027E  06 13            ASL PTR
0280  26 14            ROL PTR+1
0282  18               CLC
0283  A5 10            LDA TABLE     ;ADD POINTER AND TABLE START
0285  65 13            ADC PTR       ;ADDRESS AND PLACE RESULT IN POINTER.
0287  85 13            STA PTR
0289  A5 11            LDA TABLE+1
028B  65 14            ADC PTR+1
028D  85 14            STA PTR+1
028F  60               RTS
0290             ;
0290             ;ROUTINE TO GENERATE DATA UNIT INDEX IN TABLE
0290             ;BY HASHING 'KEY', OR CHRS OF LABEL.
0290             ;
0290  A9 00      HASH  LDA #0        ;CLEAR LOCATION FOR INDEX.
0292  18               CLC           ;PREPARE TO ADD.
0293  A0 05            LDY #5        ;LOOP 6X FOR EXCLUSIVE ORS.
0295  59 16 00   EXOR  EOR BUFFER,Y  ;EXCLUSIVE-OR ACCUM. WITH BUFFER CHR.
0298  2A               ROL A         ;MULTIPLY ACCUM. BY 2.
0299  88               DEY           ;COUNT DOWN CHRS.
029A  10 F9            BPL EXOR      ;GET NEXT CHR.
029C  85 12            STA INDX SAVE HASH PRODUCT AS INDEX.
029E  60               RTS           ;DONE.
029F                   .END

ERRORS = 0000 <0000>

SYMBOL TABLE

SYMBOL  VALUE

BAD     025D  BUFFER  0016  CHKLP  0248  CLRLP  0209
CMPR1   0227  CMPR2   0243  DECR   0211  EMPTY  0233
ENTNUM  0015  EXOR    0295  FILL   0235  FIND   023E
HASH    0290  INDX    0012  INIT   0200  LIMIT  0262
MATCH   0252  OK      026E  PTR    0013  SHADD  0272
STORE   0222  TABLE   0010  TEST   0264  XFER   0254

END OF ASSEMBLY
```

Figura 9-45 (Continuación).

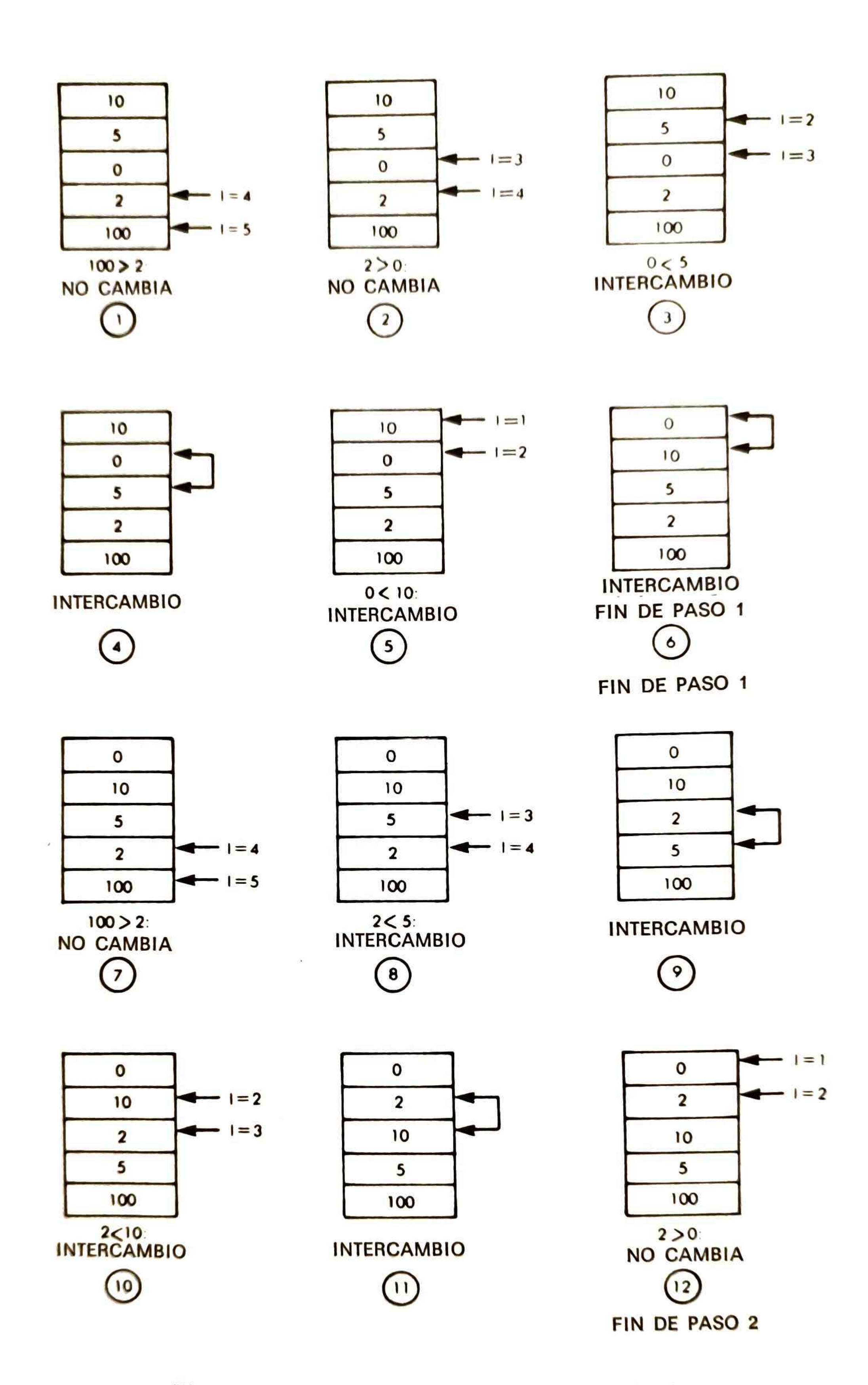


Figura 9-46 Ejemplo de clasificación de burbuja.

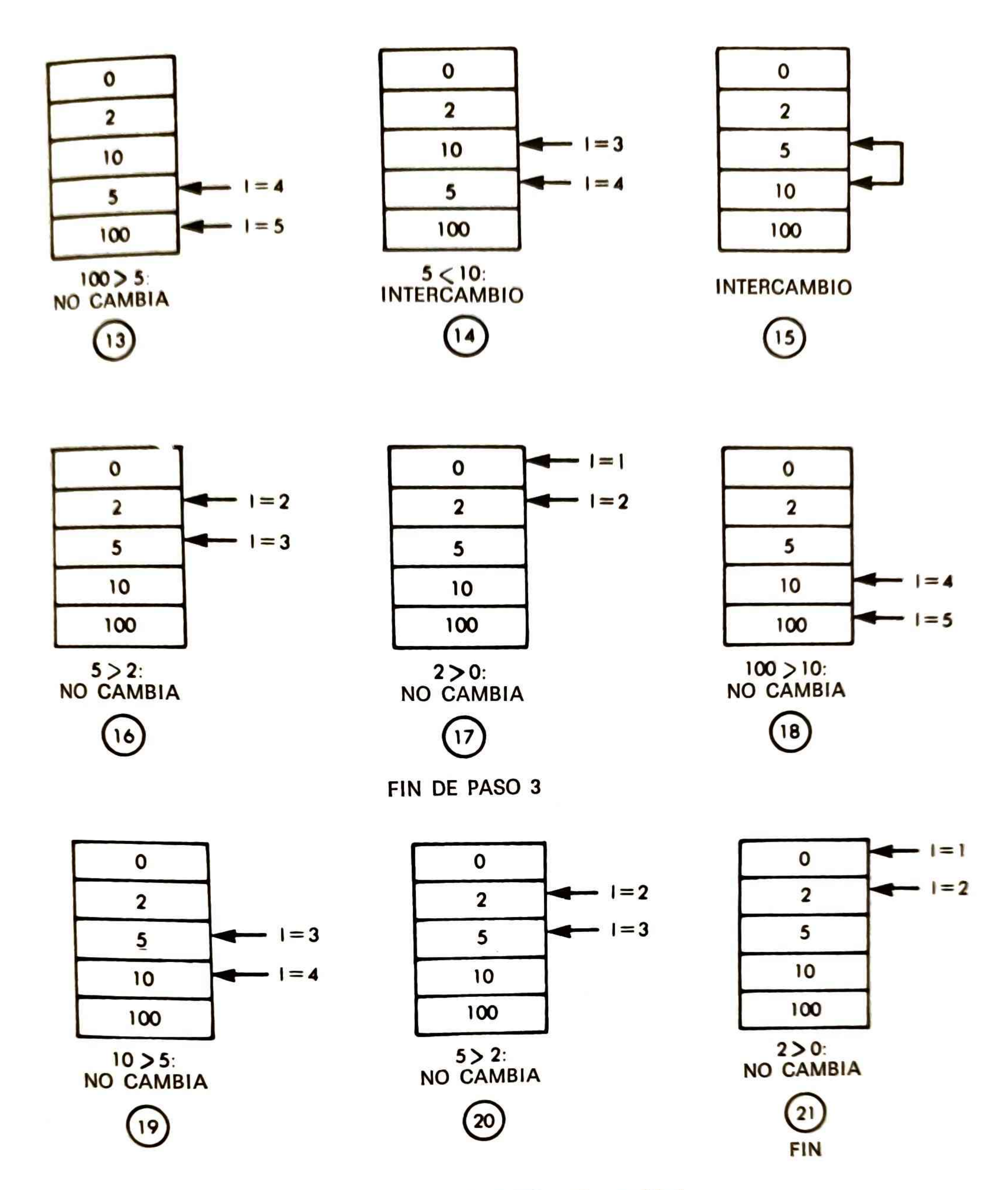


Figura 9-46 (Continuación).

Se comparan los dos elementos de arriba (o los dos de abajo). Si el inferior es menor ("más ligero") que el superior se intercambian. En caso contrario, permanecen en la misma posición. Para fines prácticos, el intercambio, si lo hubiere, se memorizará para empleo futuro. A continuación se comparará el siguiente par de elementos, etc., hasta que todos los elementos hayan sido comparados dos a dos.

El primer paso se ilustra por las etapas 1, 2, 3, 4, 5 y 6 en la figura 9-46,

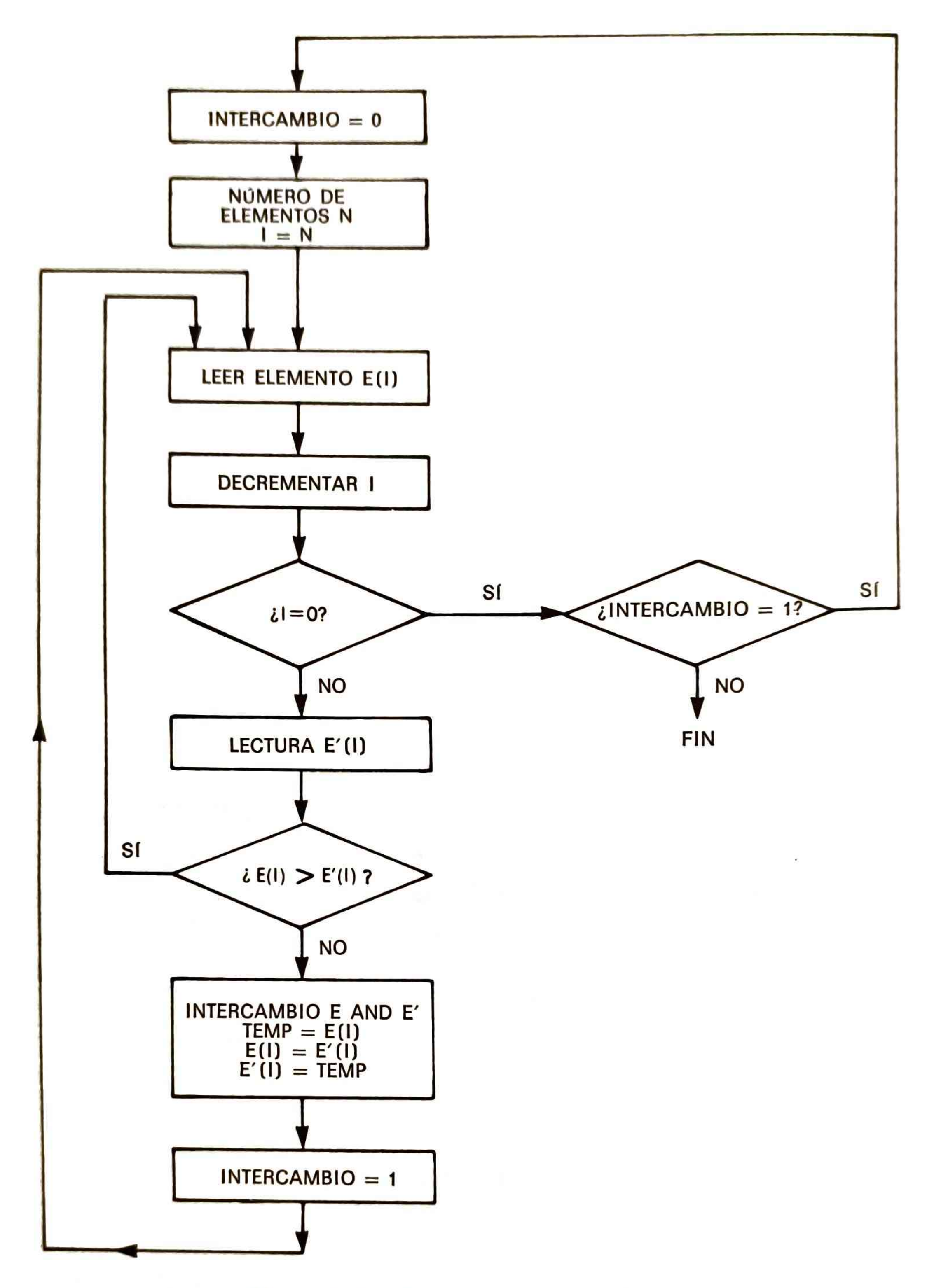


Figura 9-47 Clasificación de burbuja.

que va desde la parte inferior hacia arriba (de igual modo irá desde la parte superior hacia abajo).

Si ningún elemento ha sido intercambiado en un paso, la clasificación es completa. Si se ha producido un intercambio, se comenzará todo de nuevo.

Al examinar la figura 9-47, se puede constatar que son necesarios cuatro pasos en este ejemplo.

El proceso descrito anteriormente es sencillo y se utiliza mucho.

Una complicación adicional reside en el mecanismo real del intercambio. Cuando se intercambian A y B, no se puede escribir:

$$A = B$$
$$B = A$$

pues ello redundará en la pérdida del valor anterior de A (pruébelo con un ejemplo).

La solución correcta es utilizar una variable temporal o posición para salvaguardar el valor de A:

$$\begin{aligned} TEMP &= A \\ A &= B \\ B &= TEMP \end{aligned}$$

Ha de funcionar adecuadamente (pruebe de nuevo con un ejemplo). A esto se le llama una permutación circular y es el modo en que todos los programas

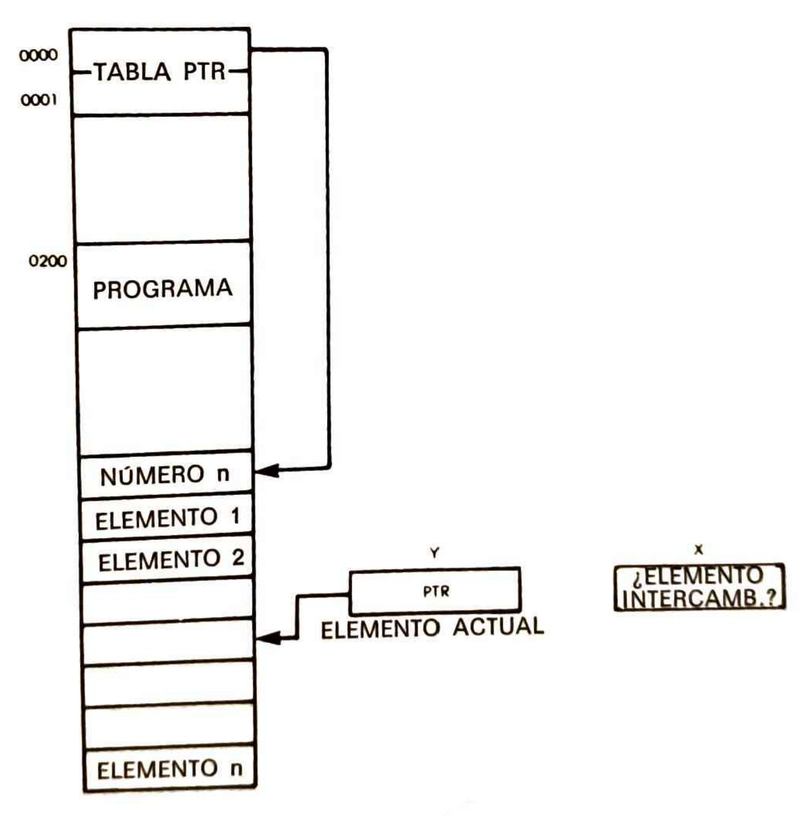


Figura 9-48 Clasificación de burbuja: mapa de memoria.

```
SORT......PAGE 0001

LINE # LOC     CODE       LINE

0002  0000                :      BUBBLE SORT PROGRAM
0003  0000                :
0004  0000                  * = $0
0005  0000                :
0006  0000  00 06         TAB    .WORD $600
0007  0002                :
0008  0002                  * = $200
0009  0200                :
0010  0200  A2 00         SORT   LDX #0          ;SET 'EXCHANGED' TO 0
0011  0202  A1 00                LDA (TAB,X)
0012  0204  A8                   TAY             ;NUMBER OF ELEMENTS IS IN Y
0013  0205  B1 00         LOOP   LDA (TAB),Y     ;READ ELEMENT E(I)
0014  0207  88                   DEY             ;DECREMENT NUMBER OF ELEMENTS TO READ.
0015  0208  F0 12                BEQ FINISH      ;END IF NO MORE ELEMENTS
0016  020A  D1 00                CMP (TAB),Y     ;COMPARE TO E'(I)
0017  020C  B0 F7                BCS LOOP        ;GET NEXT ELEMENT IF E(I)>E'(I)
0018  020E  AA            EXCH   TAX             ;EXCHANGE ELEMENTS
0019  020F  B1 00                LDA (TAB),Y
0020  0211  C8                   INY
0021  0212  91 00                STA (TAB),Y
0022  0214  8A                   TXA
0023  0215  88                   DEY
0024  0216  91 00                STA (TAB),Y
0025  0218  A2 01                LDX #1          ;SET 'EXCHANGED' TO 1
0026  021A  D0 E9                BNE LOOP        ;GET NEXT ELEMENT
0027  021C  8A            FINISH TXA             ;SHIFT 'EXCHANGED' TO A REG. FOR COMPARE...
0028  021D  D0 E1                BNE SORT        ;IF SOME EXCHANGES MADE, DO ANOTHER PASS.
0029  021F  60                   RTS
0030  0220                       .END

ERRORS = 0000 <0000>


SYMBOL TABLE

SYMBOL    VALUE

EXCH      020E   FINISH   021C   LOOP    0205   SORT    0200
TAB       0000
END OF ASSEMBLY

<
<
```

Figura 9-49 Programa de clasificación de burbuja.

realizan el intercambio. La técnica se ilustra en el diagrama de flujo de la figura 9-47.

El mapa de memoria correspondiente al programa de clasificación de burbuja se muestra en la figura 9-48. En este programa cada elemento será un número positivo de 8 bits. El programa reside en las direcciones 200 y siguientes. El registro X se utiliza para memorizar el hecho de que haya ocurrido o no un intercambio, mientras que el registro Y se utiliza como el puntero móvil en la tabla. TAB se supone está en la dirección de principio de la tabla. El programa real aparece en la figura 9-49. El direccionamiento indexado indirecto se utiliza para un acceso eficaz. Obsérvese que la clasi-

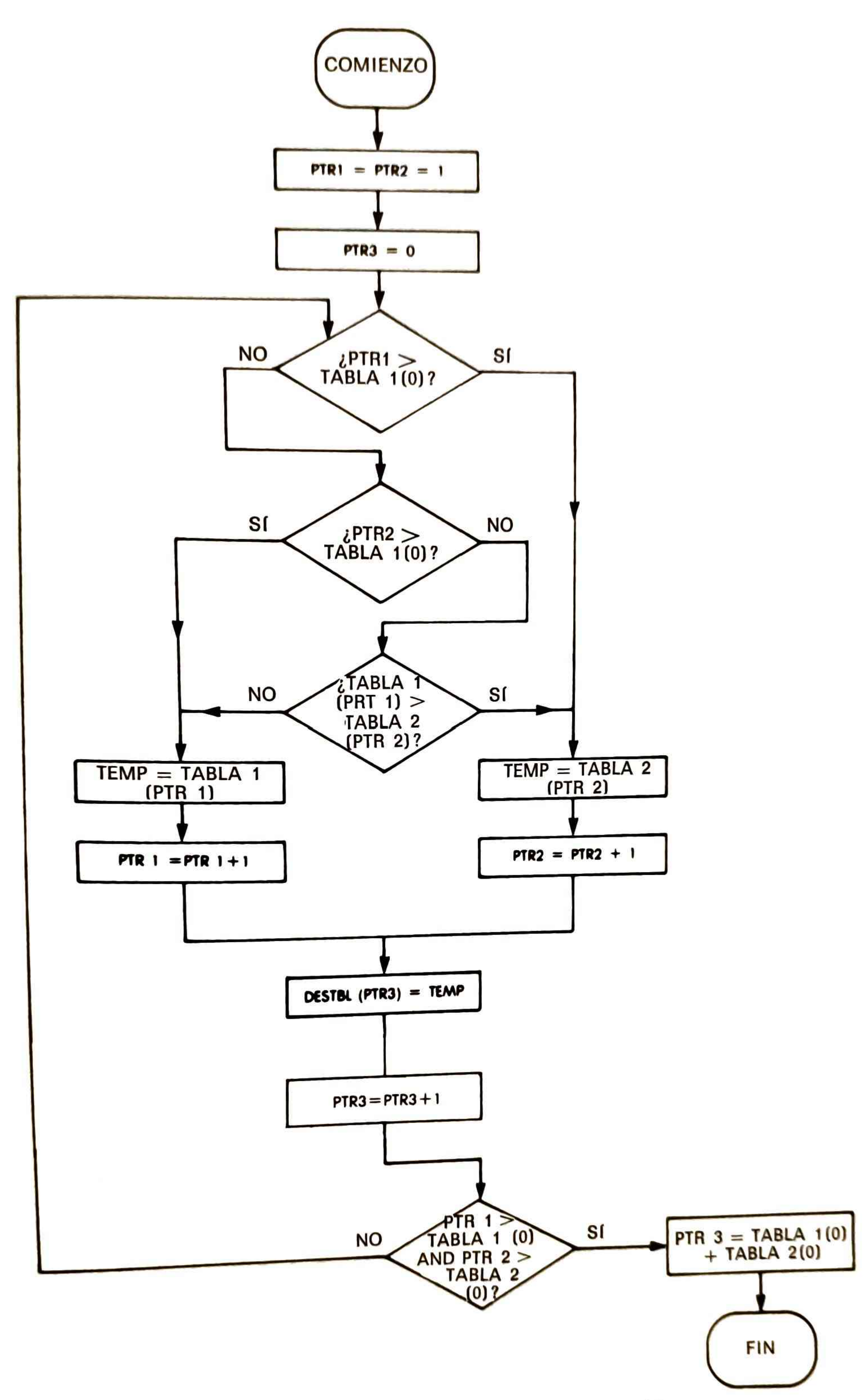


Figura 9-50 Diagrama de flujo de fusión.

ficación del programa se debe a la eficacia del modo de direccionamiento indirecto del 6502.

UN ALGORITMO DE FUSIÓN

Otro problema frecuente consiste en fusionar dos conjuntos de datos en un tercero. Supondremos que hay dos tablas de datos que han sido clasificadas previamente y deseamos fusionarlas en una tercera tabla. La longitud de cada una de las dos tablas originales se limitará a 256 bytes (una página). La primera entrada de cada tabla contiene la longitud de la misma.

El algoritmo de fusión de dos tablas se muestra en la figura 9-50. La organización de memoria correspondiente se muestra en la figura 9-51 y el programa en la figura 9-52. Recuerde posicionar "TABLE 1", "TABLE 2" y "DESTBL" antes de utilizar el programa.

El algoritmo en sí mismo es sencillo. Los punteros móviles PTR1 y PTR2 apuntan a las dos tablas de fuente. PTR3 apunta a la tabla resultado.

Las entradas reales de la TABLA 1 y TABLA 2 se comparan las dos a la vez. La más pequeña se copia en la TABLA 3 y se incrementa el correspondiente puntero móvil. El proceso se repite y termina cuando PTR1 y PTR2 han alcanzado la parte inferior de sus tablas respectivas.

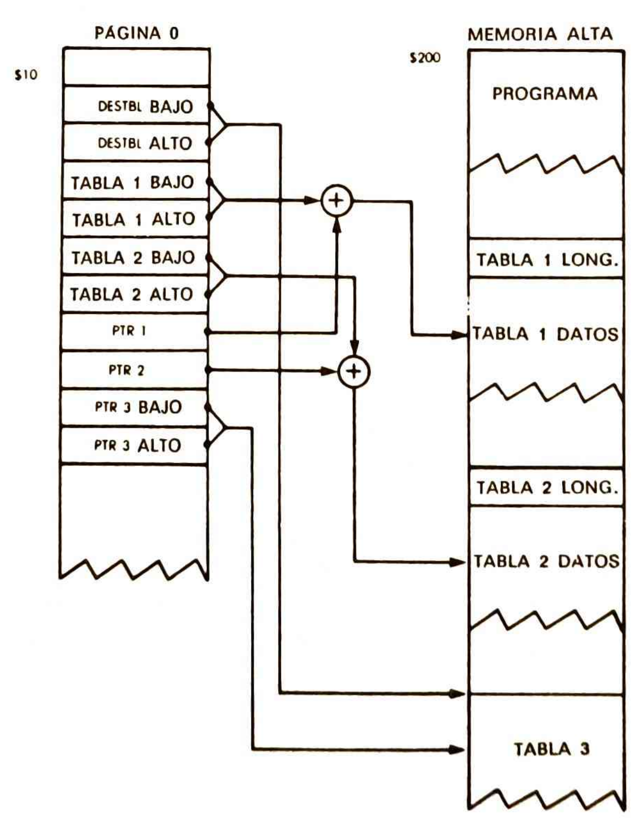


Figura 9-51 Mapa de memoria de fusión.

```
LOC     CODE        LINE

0000                ;2-PAGE MERGE.
0000                ;TAKES 2 DATA TABLES PREVIOUSLY SORTED,
0000                ;AND MERGES THEM INTO A THIRD TABLE.
0000                ;EACH SOURCE TABLE CAN BE UP TO ONE
0000                ;PAGE (256 BYTES) IN LENGTH.
0000                ;THE FIRST ELEMENT OF THE SOURCE
0000                ;TABLES MUST CONTAIN THE TABLE LENGTH.
0000                ;'PTR3' CONTAINS THE LENGTH OF THE
0000                ;DESTINATION TABLE AT RETURN.
0000                ;
0000                        * = $10
0010                DESTBL  *=*+2           ;POINTER TO BEGINNING OF DESTINATION TABLE.
0012                TABLE1  *=*+2           ;POINTER TO SOURCE TABLE 1.
0014                TABLE2  *=*+2           ;POINTER TO SOURCE TABLE 2.
0016                PTR1    *=*+1           ;TABLE 1 INDEX.
0017                PTR2    *=*+1           ;TABLE 2 INDEX.
0018                PTR3    *=*+2           ;DESTINATION TABLE INDEX.
001A                ;
001A                        * = $200
0200                ;
0200    A5 11               LDA DESTBL+1    ;PTR3 = TABLE3
0202    85 19               STA PTR3+1
0204    A5 10               LDA DESTBL
0206    85 18               STA PTR3
0208    A9 01               LDA #1          ;SET SOURCE TABLE POINTERS TO BEGINNING,
020A    85 16               STA PTR1        ;SKIPPING TABLE LENGTHS.
020C    85 17               STA PTR2
020E    A2 00               LDX #0          ;CLEAR X FOR INDIRECT ADDRESSING.
0210    A1 14       COMPR   LDA (TABLE2,X)  ;IS TABLE 2 LENGTH <
0212    C5 17               CMP PTR2        ;TABLE 2 POINTER?
0214    90 19               BCC TKTB1       ;IF YES, GET BYTE FROM TABLE 1.
0216    A1 12               LDA (TABLE1,X)  ;IS TABLE 1 LENGTH <
0218    C5 16               CMP PTR1        ;TABLE 1 POINTER?
021A    90 0A               BCC TKTB2       ;IF YES, GET BYTE FROM TABLE 2
021C    A4 16               LDY PTR1        ;GET POINTER FOR TABLE 1.
021E    B1 12               LDA (TABLE1),Y  ;USE IT TO FETCH BYTE.
0220    A4 17               LDY PTR2        ;GET POINTER FOR TABLE 2,
0222    D1 14               CMP (TABLE2),Y  ;USE IT TO FIND BYTE TO COMPARE
0224                         ;TO TABLE 1 BYTE.
0224    90 09               BCC TKTB1       ;IF TABLE 1 BYTE LESS, TAKE IT.
0226    A4 17       TKTB2   LDY PTR2        ;GET POINTER FOR TABLE 2.
0228    B1 14               LDA (TABLE2),Y  ;GET NEXT BYTE FROM TABLE 2.
022A    E6 17               INC PTR2        ;INCREMENT POINTER FOR TABLE 2.
022C    4C 35 02            JMP STORE       ;GO STORE BYTE IN DESTINATION TABLE.
022F    A4 16       TKTB1   LDY PTR1        ;GET POINTER 1...
0231    B1 12               LDA (TABLE1),Y  ;AND USE IT TO GET BYTE FROM TABLE.
0233    E6 16               INC PTR1        ;INCREMENT POINTER FOR TABLE 1.
0235    81 18       STORE   STA (PTR3,X)    ;STORE BYTE AT NEXT LOCATION IN TABLE 3
0237    E6 18               INC PTR3        ;INCREMENT LO ORDER TABLE 3 POINTER.
0239    D0 02               BNE CC          ;IF NO OVERFLOW, SKIP
023B    E6 19               INC PTR3+1      ;INCREMENT HI ORDER TABLE 3 POINTER.
023D    A1 12       CC      LDA (TABLE1,X)  ;IS TABLE 1 LENGTH GREATER
023F    C5 16               CMP PTR1        ;THAN OR EQUAL TO POINTER 1?
0241    B0 CD               BCS COMPR       ;IF YES, GET NEXT BYTE.
0243    A1 14               LDA (TABLE2,X)  ;IS TABLE 2 LENGTH GREATER
0245    C5 17               CMP PTR2        ;THAN OR EQUAL TO POINTER 2?
0247    B0 C7               BCS COMPR       ;IF YES, GET NEXT BYTE.
0249    A9 00               LDA #0
024B    85 19               STA PTR3+1      ;CLEAR PTR3 HI ORDER.
024D    18                  CLC             ;MERGE DONE, NOW..
024E    A1 12               LDA (TABLE1,X)  ;ADD TABLE 1 AND 2 LENGTHS.
0250    61 14               ADC (TABLE2,X)
0252    85 18               STA PTR3        ;STORE SUM IN TABLE 3 TEMPORARY POINTER.
0254    90 04               BCC CCC         ;AND..
0256    A9 01               LDA #1          ;OVERFLOW IN...
0258    85 19               STA PTR3+1      ;HI BYTE.
025A    60          CCC     RTS
025B                        .END

ERRORS = 0000 <0000>
END OF ASSEMBLY
```

Figura 9-52 Programa de fusión.

RESUMEN

Se han presentado ejemplos de realización reales, así como los conceptos básicos relativos a estructuras de datos comunes.

Debido a sus potentes modos de direccionamiento, el 6502 se presta bien a la gestión de estructuras de datos complejos. Su eficacia se demuestra por la brevedad de los programas mostrados.

Además se han presentado técnicas especiales de elecciones aleatorias, clasificación y fusión, que son típicas de los requeridos para solucionar problemas complejos que lleven consigo estructuras de datos reales.

El programador principiante podrá despreocuparse de los detalles de la realización de las estructuras de datos y su gestión. Sin embargo, para la programación eficaz de algoritmos no triviales, se requiere una buena comprensión de las estructuras de datos. Los programas reales presentados en este capítulo ayudarán al lector a alcanzar tal comprensión y solucionar todos los problemas habituales a estructuras de datos reales.

10 Desarrollo de los programas

INTRODUCCIÓN

Todos los programas que hemos estudiado y desarrollado hasta ahora se han desarrollado a mano, sin la ayuda de ningún recurso hardware o software. La única mejora que hemos introducido con relación al código puramente binario ha sido el empleo de símbolos nemónicos, los del lenguaje ensamblador. Para un desarrollo eficaz del software es necesario comprender la gama de ayudas al desarrollo de hardware y software de las que se dispone. El objetivo de este capítulo es presentar y evaluar estas ayudas.

ELECCIÓN FUNDAMENTAL DE LA PROGRAMACIÓN

Existen tres alternativas fundamentales: escritura de un programa en binario o hexadecimal, escribirlo en lenguaje ensamblador, o escribirlo en lenguaje de alto nivel o evolucionado. Veamos estas posibilidades.

1. Codificación hexadecimal

El programa se escribirá normalmente con el empleo de los nemónicos del lenguaje ensamblador. Sin embargo, la mayoría de los sistemas de ordenador de bajo coste no poseen ensamblador. El ensamblador es el programa que convertirá automáticamente los nemónicos utilizados por el programa en los códigos binarios requeridos. Cuando no se dispone de ensamblador, esta conversión de nemónicos en binario se debe efectuar a mano. El binario es incómodo de utilizar y propicio a los errores, por lo que se suele emplear

el hexadecimal. Se ha visto en el capítulo 1 que un dígito hexadecimal representa 4 bits en binario. Para representar el contenido de cada byte se utilizarán, pues, dos dígitos hexadecimales. A título de ejemplo, una tabla que da el equivalente hexadecimal de las instrucciones del 6502 figura en el apéndice.

Cuando los recursos financieros del usuario están limitados y ningún ensamblador está disponible, tendrá que convertir el programa en hexadecimal a mano. Ello se puede hacer para un número pequeño de instrucciones, tales como pueden ser de 10 a 100. Para programas más largos, este proceso es tedioso y propicio a errores, por lo que ha de tratar de evitarlo. Sin embargo, casi todos los microordenadores de una sola tarjeta necesitan entrada de programas en modo hexadecimal. No poseen ensamblador y, con el fin de limitar su coste, no están provistos de teclado alfanumérico completo.

En resumen, la codificación hexadecimal no es una manera ideal de introducir un programa en un ordenador. Es sencillamente un medio económico. Se establece una solución de compromiso coste/eficacia entre el precio de la configuración necesaria para el empleo de un ensamblador y el del teclado alfanumérico requerido, y la tarea fastidiosa de introducir el programa en la memoria en hexadecimal. Sin embargo, ello no cambia el modo en que se escribe el programa propiamente dicho. El programa se sigue escribiendo en lenguaje de nivel ensamblador, con el fin de que no solamente pueda ser significativo, sino que también pueda ser fácilmente examinado por el programador.

2. Programación en lenguaje ensamblador

La programación en nivel ensamblador cubre programas que se pueden introducir en hexadecimal, así como los que se pueden introducir en forma de ensamblador simbólico en el sistema. Examinemos ahora la entrada de un programa directamente en su representación en lenguaje ensamblador. Supondremos que se dispone de un programa ensamblador. El ensamblador leerá cada una de las instrucciones nemónicas del programa y las convertirá en el modelo binario requerido, utilizando 1, 2 o 3 bytes, como lo especifica la codificación de las instrucciones. Además, un buen ensamblador ofrece ciertas facilidades para escribir el programa. Éstas se revisarán en la sección dedicada al ensamblador. En particular, hay *directivos* (pseudoinstrucciones) disponibles que modificarán el valor de los símbolos. El direccionamiento simbólico se puede utilizar y se puede efectuar una bifurcación a una posición simbólica. Durante la fase de depuración, en la que un usuario puede eliminar o añadir instrucciones, no se necesitará reescribir el programa completo, si se inserta una instrucción adicional entre una bifurcación y el punto

en que bifurca, en tanto que se utilicen las etiquetas simbólicas. El ensamblador ajustará automáticamente todas las etiquetas durante el proceso de conversión. Además, un ensamblador permite al usuario depurar su programa en forma simbólica. Se puede utilizar un *desensamblador* para examinar el contenido de una posición de memoria y reconstruir la instrucción de nivel ensamblador que representa. A continuación se revisarán los diferentes recursos software de los que suele disponerse en un sistema. Examinemos la tercera posibilidad de programación (fig. 10-1).

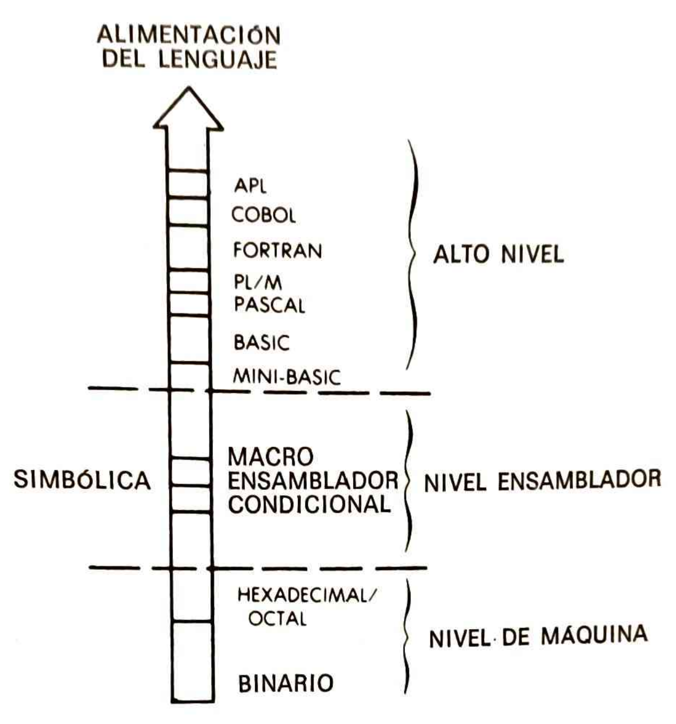


Figura 10-1 Niveles de programación.

3. Lenguaje de alto nivel

Un programa se puede escribir en un lenguaje de alto nivel tal como BASIC, APL, PASCAL u otros. Las técnicas de programación en estos diversos lenguajes se exponen en libros específicos y no se tratarán en este libro. En consecuencia, sólo se verá brevemente este modo de programación. Un lenguaje de alto nivel ofrece instrucciones potentes que hacen la programación mucho más fácil y rápida. Estas instrucciones se deben traducir, por un programa complejo, en la representación binaria definitiva que sólo puede ejecutar un microordenador. En general, cada instrucción de alto nivel se convertirá en un número grande de instrucciones binarias elementales. El programa que efectúa esta conversión automática se llama *compilador o*

intérprete. Un compilador convertirá todas las instrucciones secuenciales de un programa en código objeto. El código resultante se ejecuta entonces en una fase distinta. Por el contrario, un intérprete interpretará una sola instrucción a la vez y la ejecuta, después "convierte" la siguiente y la ejecuta. Un intérprete ofrece la ventaja de una respuesta interactiva, pero resulta menos eficaz que un compilador en lo que respecta a la velocidad de ejecución. Estos temas no se estudiarán posteriormente. Volvamos a la programación de un microprocesador real en lenguaje ensamblador.

APOYO SOFTWARE

Veamos, los principales medios que están (o deben estar) disponibles en el sistema completo para un desarrollo software adecuado. Algunos de los programas que ya han sido introducidos y sus definiciones se resumirán a continuación. Las definiciones de otros programas importantes se proporcionarán también antes de proseguir.

El *ensamblador* es el programa que convierte (traduce) la representación nemónica de las instrucciones en su equivalente binario. Suele convertir una instrucción simbólica en una instrucción binaria (que puede ocupar 1, 2 o 3 bytes). El código binario resultante se llama *código objeto.* Es ejecutable directamente por el microordenador. Como subproducto, el ensamblador proporcionará también un listado simbólico completo del programa, así como las tablas de equivalencia a utilizar por el programador y la lista de ocurrencia de los símbolos en el programa. Se presentarán ejemplos más adelante en este mismo capítulo.

Un *compilador* es un programa que convierte las instrucciones del lenguaje de alto nivel a su forma binaria.

Un *intérprete* es un programa similar a un compilador. Convierte también las instrucciones de alto nivel en su forma binaria, pero en vez de conservar las representaciones intermedias, ejecuta las instrucciones inmediatamente, De hecho, y con frecuencia, no genera un código intermedio, sino que ejecuta las instrucciones de alto nivel directamente.

Un *monitor* es un programa base indispensable para utilizar los recursos hardware del sistema. Vigila continuamente los dispositivos de entrada y controla el resto de los dispositivos. Por ejemplo, un monitor mínimo de un microordenador en una sola tarjeta, provisto de un teclado y con varios LED, debe explorar continuamente el teclado para detectar entradas del usuario y visualizar los datos deseados mediante los diodos emisores de luz. Además, debe ser capaz de comprender un número limitado de mandatos desde el teclado, tales como START, STOP, CONTINUE, LOAD MEMORY y EXAMINE MEMORY. En un sistema grande, el monitor se conoce, en

ocasiones, como el programa *ejecutivo,* cuando se proporciona también una gestión de ficheros compleja o tareas de planificación (previsión de tiempos). Al conjunto completo de recursos se le llama *sistema operativo.* En el caso de que residan los ficheros en un disco, el sistema operativo se denomina *sistema operativo de disco* o DOS ("disk operating system").

Un *editor* es el programa concebido para facilitar la entrada y modificación de textos, o programas. Permite al usuario introducir caracteres cómodamente, añadirlos, insertarlos, añadir líneas, eliminar líneas y buscar caracteres o grupos. Es un recurso importante para introducir textos de modo adecuado y efectivo.

Un *depurador* ("debugger") es un recurso necesario para la depuración (corrección o puesta a punto) de programas. Cuando un programa no funciona correctamente, puede que no haga indicación sobre la causa de tal circunstancia. El programador, en consecuencia, debe insertar puntos de ruptura (break-points) en su programa para interrumpir la ejecución del programa en la dirección especificada y poder examinar el contenido de registros o memoria en estos puntos. Esta es la función principal de un depurador. El depurador permite pues la posibilidad de interrumpir un programa, reanudar la ejecución, examinar, visualizar y modificar el contenido de registros o memoria. Un buen depurador debe estar provisto de un número de recursos adicionales, tales como la posibilidad de examinar datos en forma simbólica, hexadecimal, binaria u otra representación habitual, así como introducir datos en este formato.

Un *cargador o cargador editor de enlaces* ("loader" o "linking loader") situará varios bloques de código objeto en las posiciones especificadas de la memoria y ajustará sus punteros simbólicos respectivos, de modo que pueden referenciarse los unos con los otros. Sirve para reubicar programas o bloques en diferentes zonas de memoria.

Un *simulador o un programa emulador* sirve para simular la operación de un dispositivo, que suele ser el microprocesador, en su ausencia, cuando se desarrolla un programa en un procesador simulado, antes de colocarse en la tarjeta definitiva. Empleando este procedimiento se hace posible interrumpir el programa, modificarlo y guardarlo en memoria RAM. Los inconvenientes de un simulador son:

1. Generalmente, sólo simula el procesador propiamente dicho, pero no los dispositivos de entrada/salida.
2. La velocidad de ejecución es pequeña y se debe trabajar en tiempo simulado. Es, por tanto, imposible comprobar dispositivos en tiempo real, lo que puede dar lugar a problemas de sincronización aunque la lógica del programa pueda encontrarse correcta.

Un *emulador* es, realmente, un simulador en tiempo real. Utiliza un procesador para simular otro y lo simula por completo y al detalle.

Las rutinas de utilidad o programas utilitarios son esencialmente todas las rutinas que el usuario desearía que el fabricante le hubiera proporcionado. Ellas pueden incluir multiplicación, división y otras operaciones aritméticas, rutinas de transferencia de bloques, comprobación de caracteres, rutinas de gestión de dispositivos de entrada/salida, etc.

LA SECUENCIA DE DESARROLLO DEL PROGRAMA

Examinaremos, ahora, una secuencia típica para el desarrollo de un programa al nivel ensamblador. Para poner de manifiesto su valor supondremos que están disponibles todos los recursos software habituales. Si no fuera así en un sistema dado, será posible desarrollar programas, pero disminuyendo la comodidad y, en consecuencia, es probable que aumente la magnitud de tiempo que se necesita para depurar el programa.

El procedimiento normal es concebir, en primer lugar, un algoritmo y definir las estructuras de datos adecuadas al problema a resolver. A continuación se desarrolla un conjunto comprensible de diagramas de flujo que representen el programa. Finalmente, los diagramas de flujo se traducen a lenguaje de nivel ensamblador para el microprocesador; esta es la fase de codificación.

Seguidamente, se tiene que introducir el programa en el ordenador. Examinaremos en la sección siguiente las opciones hardware que se utilizan en esta fase.

El programa se introduce en la memoria RAM del sistema bajo el control del editor. Cuando se ha introducido una sección del programa, tal como una subrutina, deberá ser objeto de comprobación.

En primer lugar se utilizará el ensamblador. Si el mismo no reside ya en el sistema, se cargará desde una memoria externa, tal como un disco. A continuación será ensamblado el programa, es decir, convertido en un código binario. Resulta de ello un programa objeto preparado para ser ejecutado.

No se debe esperar normalmente que un programa funcione correctamente la primera vez. Para verificar su funcionamiento correcto, suele establecerse un cierto número de puntos de ruptura en posiciones cruciales, en donde es fácil comprobar si los resultados intermedios son correctos. El depurador se utilizará para este propósito. Los puntos de ruptura se especificarán en posiciones seleccionadas (direcciones). Se enviará, por ello, un mandato "Go", de modo que comience la ejecución del programa. El programa se detendrá automáticamente en cada uno de los puntos de ruptura especi-

ficados. El programador puede entonces verificar que los datos hasta este punto son correctos, examinando el contenido de los registros, o de la memoria. Si son correctos, continuaremos hasta el siguiente punto de ruptura. Siempre que encontremos datos incorrectos, se ha identificado un error en el programa. En este momento, el programador suele referirse a su listado del programa y verificar si su codificación ha sido correcta. Si ningún error puede ser encontrado en la programación, se puede tratar de un error lógico que se refiere, de nuevo, al diagrama de flujo. En este caso se supondrá razonablemente que los diagramas de flujo han sido comprobados a mano y que son correctos. El error probablemente tenga su causa en la codificación. Será necesario, pues, modificar una parte del programa. Si la representación simbólica del programa sigue estando en la memoria, se debe reintroducir simplemente el editor y modificar las líneas requeridas y, a continuación, volver de nuevo a la secuencia precedente. En ciertos sistemas, la memoria disponible puede no ser suficiente, de modo que es necesario vaciar la representación simbólica del programa en un disco o cassette, antes de ejecutar el código objeto. Naturalmente, en tal caso, se tendrá que volver a cargar la representación simbólica del programa a partir de su medio de soporte, antes de introducir de nuevo el editor.

El procedimiento anterior se repetirá mientras sea necesario, hasta que los resultados del programa sean correctos. Insistimos en el hecho de que es mucho más efectivo prevenir que curar. Una concepción correcta conducirá, generalmente, a un programa que funcionará correctamente de modo muy rápido una vez que se han eliminado los errores de escritura habituales o los errores de codificación evidentes. Sin embargo, una concepción limpia puede dar lugar a programas cuya depuración llevará mucho tiempo. El tiempo de depuración suele considerarse mucho mayor que el tiempo de concepción propiamente dicho (normalmente unas diez veces). En resumen, es siempre más rentable invertir más tiempo en la concepción con el fin de acortar la fase de depuración.

Aunque el empleo de este método permite comprobar la organización global del programa, no se presta por sí mismo a comprobar el programa en términos de tiempo real y con dispositivos de entrada/salida. Si los dispositivos de entrada/salida se tienen que comprobar, la solución directa consiste en transferir el programa a memorias EPROM e instalarlas en la tarjeta y luego comprobar si funciona.

Existe una solución mejor, que es el empleo de un *emulador en circuito*. Dicho dispositivo utiliza el microprocesador 6502 (o cualquier otro microprocesador) para emular un 6502 en tiempo (casi) real. Emula físicamente al 6502. El emulador está provisto de un cable que termina en un conector de 40 patillas, exactamente igual patilla a patilla que el 6502. Este conector puede insertarse en la tarjeta de aplicación real que se está desarro-

llando. Las señales generadas por el emulador serán exactamente las del 6502, con la salvedad de que serán un poco más lentas. La ventaja esencial es que el programa objeto de prueba residirá todavía en la RAM del sistema de desarrollo. Generará las señales que comunicarán con los dispositivos de entrada/salida reales que desea utilizar. Como resultado, se hace posible continuar el desarrollo del programa utilizando los recursos del sistema de desarrollo (editor, depurador, recursos simbólicos, sistema de ficheros) mientras se comprueban las entradas/salidas en tiempo real.

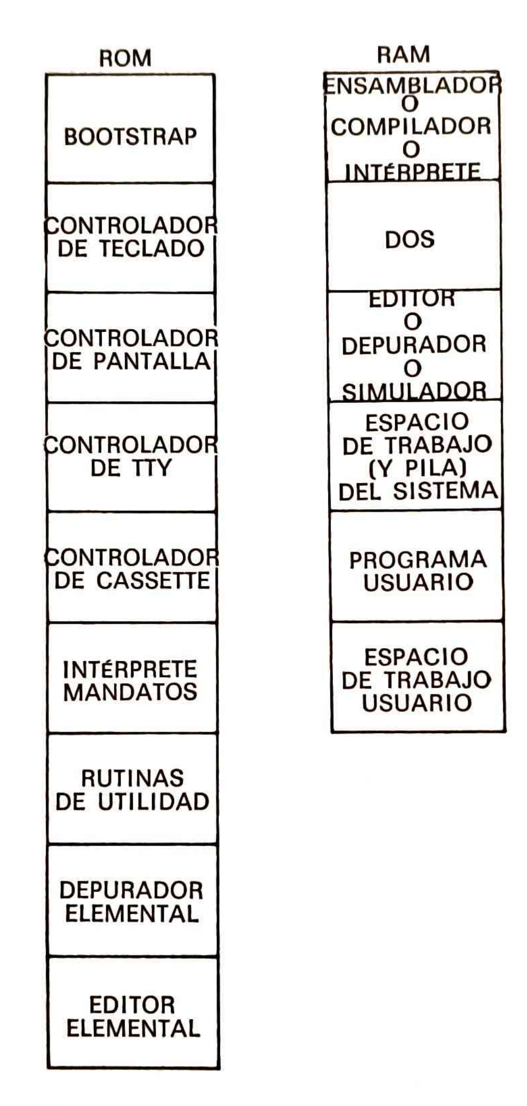


Figura 10-2 Mapa de memoria tipo.

Además, un buen emulador ofrece posibilidades especiales, tales como el *rastreo* ("trace"). Esta posibilidad consiste en un registro de las últimas instrucciones o del estado de los diferentes buses de datos del sistema antes de un punto de ruptura. En resumen, proporciona la "película" de los sucesos

que se produjeron antes del punto de ruptura o del mal funcionamiento. Puede incluso disparar un osciloscopio en una dirección específica o al ocurrir una combinación determinada de bits. Tal recurso es de un gran valor, ya que cuando se encuentra un error suele ser demasiado tarde. La instrucción o los datos que produjeron el error, se han producido antes de la detección. La disponibilidad de un análisis de rastreo permite al usuario encontrar qué parte del programa ha originado el error. Si el rastreo no es demasiado largo, se puede establecer simplemente un punto de ruptura anterior.

Lo anterior completa nuestra descripción de la secuencia habitual de acontecimientos implicados en el desarrollo de un programa. Veamos ahora las posibilidades de hardware disponibles para el desarrollo de programas.

LAS ALTERNATIVAS DE HARDWARE

1. Microordenador en una sola tarjeta

El microordenador de una sola tarjeta permite los recursos de más bajo coste para el desarrollo de programas. Suele estar provisto de un teclado hexadecimal, algunas teclas de función y 6 LED que pueden visualizar las direcciones y los datos correspondientes. Como está dotado de una memoria pequeña, no suele disponer de ningún ensamblador. A lo sumo tiene un pequeño monitor y ninguna posibilidad de edición o depuración, salvo para unos pocos mandatos. Todos los programas, por tanto, se deben introducir en formato hexadecimal. Se visualizarán también en formato hexadecimal en los LED. Un microordenador de una sola tarjeta tiene, en teoría, la misma potencia hardware que cualquier otro ordenador. Sin embargo, como su capacidad de memoria y su teclado son limitados, no tienen todos los recursos usuales de un sistema más grande y esto hace el programa de desarrollo mucho más largo. La incomodidad del desarrollo de programas en formato hexadecimal hace que los microordenadores de una sola tarjeta sean más adecuados para aplicaciones de enseñanza y de aprendizaje, en donde es deseable la reducción de su longitud. Las tarjetas autónomas son probablemente el medio más económico para el aprendizaje activo de la programación. No obstante, ellas no se pueden utilizar para el desarrollo de programas complejos, a no ser que se conecten a tarjetas de memoria adicionales y se disponga de las ayudas software habituales.

2. El sistema de desarrollo

Un sistema de desarrollo es un sistema de microordenador dotado de una capacidad significativa de memoria RAM (32-48K) así como de los

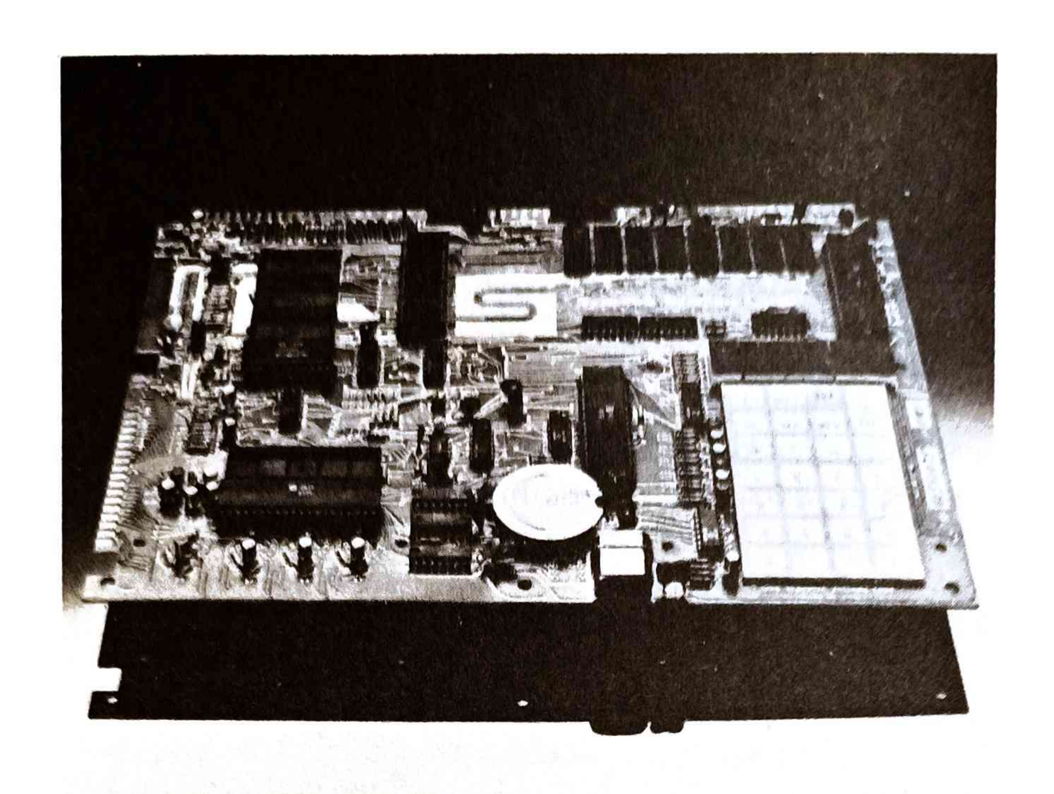


Figura 10-3 El SYM 1 es un microordenador típico en una tarjeta.

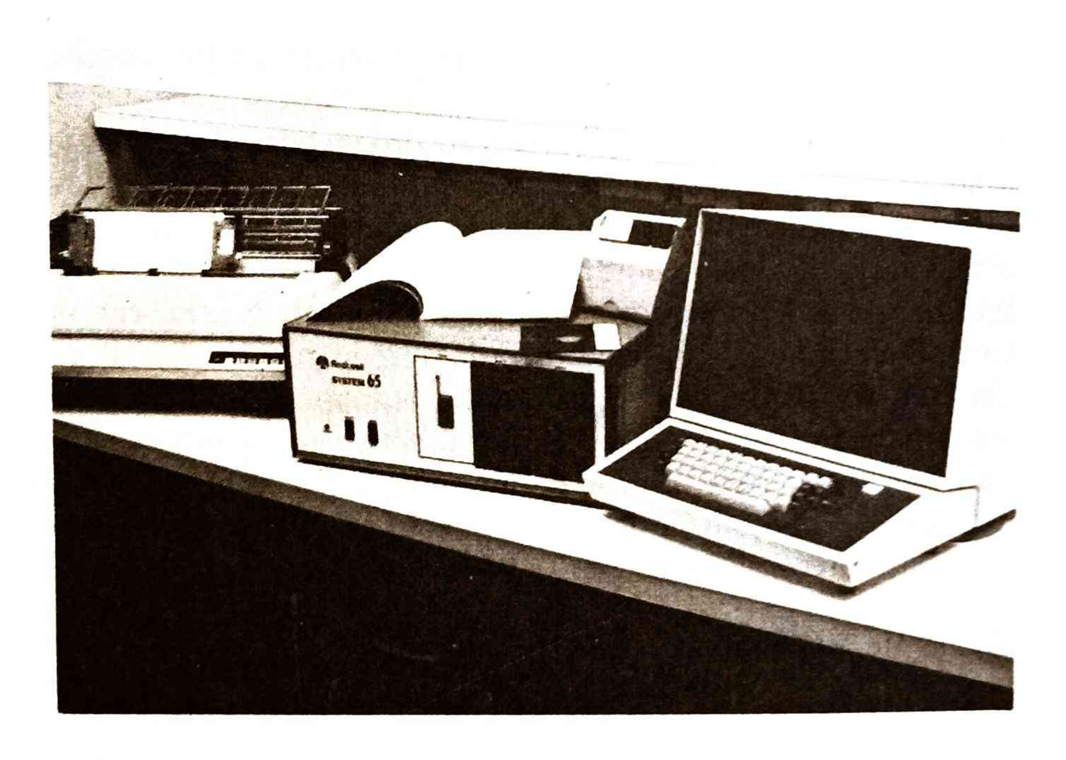


Figura 10-4 El sistema 65 de Rockwell es un sistema de desarrollo.

dispositivos necesarios de entrada/salida, tales como una pantalla de visualización (TRC), una impresora, discos y, habitualmente, un programador de PROM, así como, a veces, un emulador en circuito. Un sistema de desarrollo está concebido específicamente para facilitar el desarrollo del programa en una aplicación industrial. Normalmente ofrece todos, o casi todos, los recursos que se han mencionado en la sección anterior. En principio, es la herramienta ideal para el desarrollo de software.

La limitación de un sistema de desarrollo de microordenador es que no puede ser capaz de tener un compilador o un intérprete. Ello se debe a que un compilador suele exigir una gran capacidad de memoria, con frecuencia mayor que la disponible en el sistema. Sin embargo, para el desarrollo de programas en lenguaje de nivel ensamblador, el sistema de desarrollo ofrece todos los recursos necesarios. Lamentablemente, como los sistemas de desarrollo se venden en un número relativamente pequeño comparado con los ordenadores individuales, su coste es considerablemente más alto.

3. Microordenadores individuales

El hardware de un microordenador individual es análogo al de un sistema de desarrollo. La diferencia principal radica en el hecho de que el microordenador individual no suele estar dotado de las ayudas al desarrollo del software complejas que están disponibles en un sistema de desarrollo industrial. Por ejemplo, muchos microordenadores individuales ofrecen solamente ensambladores elementales, editores mínimos, sistemas de ficheros mínimos, pero no tienen posibilidad de conectarse a un programador PROM, ni a un emulador en circuito ni a un depurador potente. Representan, pues, un paso intermedio entre el microordenador de una sola tarjeta y el sistema de desarrollo completo. Para un usuario que desea desarrollar programas de complejidad moderada, constituyen probablemente la mejor solución de compromiso, puesto que ofrecen la ventaja de un coste bajo y un conjunto razonable de herramientas de desarrollo software, aunque son bastante limitadas desde el punto de vista de la comodidad del empleo.

4. Sistemas de tiempo compartido

Diferentes compañías alquilan terminales que pueden ser conectados a redes de ordenador de tiempo compartido. Estos terminales comparten el tiempo de los grandes ordenadores y se benefician de todas las ventajas de las grandes instalaciones. *Los ensambladores cruzados* están disponibles para todos los microordenadores en, prácticamente, todos los sistemas comerciales de tiempo compartido. Un ensamblador cruzado es simplemente un ensamblador para, digamos, el 6502, que reside, por ejemplo, en un IBM370.

Formalmente, un ensamblador cruzado es un ensamblador para el microprocesador X que reside en el procesador Y. La naturaleza del ordenador que se utilice carece de importancia. El usuario escribe siempre un programa en el lenguaje de nivel ensamblador del 6502, y el ensamblador cruzado lo convierte en el modelo binario adecuado. La única dificultad radica en el hecho que este programa no puede ser ejecutado inmediatamente. Puede ser ejecutado por un procesador simulador, si se dispone del mismo, pero solamente si el programa no utiliza ningún recurso de entrada/salida. Debido a este inconveniente, por tanto, el tiempo compartido solamente es práctico en aplicaciones industriales.

5. Ordenador local (in situ)

Siempre que se dispone de un gran ordenador local, se pueden también utilizar los ensambladores cruzados para facilitar el desarrollo de un programa. Si tal ordenador ofrece un servicio de tiempo compartido, esta opción es esencialmente análoga a la anterior. Si sólo se ofrece servicio por lotes ("batch"), es probablemente uno de los métodos más incómodos del desarrollo de programas, ya que la presentación de programas en el modo de "lotes" en el nivel ensamblador para un microprocesador da lugar a un tiempo de desarrollo muy largo.

¿Es necesario el panel frontal de control?

El panel frontal es un accesorio hardware que suele emplearse para facilitar la depuración del programa. Ha sido la herramienta tradicional para visualizar cómodamente el contenido binario de un registro, o de memoria. Sin embargo, la mayoría de las funciones del panel de control pueden ser realizadas desde un terminal a través de una pantalla TRC y esta última permite visualizar el valor binario de los bits, por lo que ofrece un servicio casi equivalente al del panel de control. La ventaja adicional de utilizar la pantalla TRC es que se puede pasar, a voluntad, desde la representación binaria a hexadecimal, simbólica, decimal (naturalmente, si se dispone de las rutinas de conversión adecuadas). El inconveniente principal del TRC es que, en vez de girar un botón, se deben pulsar diferentes teclas para obtener la visualización deseada.

Sin embargo, como el coste de un panel de control es bastante importante, la mayoría de los microordenadores modernos han abandonado esta herramienta de depuración en favor del TRC. La importancia del panel de control suele evaluarse más en función de argumentos emocionales basados en su propia experiencia que en función de una elección racional. Por ello, no es indispensable.

RESUMEN DE LOS RECURSOS HARDWARE

Se deben distinguir tres categorías de tarjetas. Si dispone solamente de un presupuesto limitado y desea aprender a programar, compre un microordenador de una sola tarjeta. Su empleo podrá desarrollar todos los programas sencillos de este libro y muchos más. Por el contrario, cuando se desee desarrollar programas de más de un centenar de instrucciones, se notarán las limitaciones de este método.

Un usuario industrial necesitará un sistema de desarrollo completo. Cualquier solución abreviada del sistema de desarrollo completo conducirá a tiempos de desarrollo más grandes. El compromiso es evidente: recursos hardware contra tiempos de programación. Naturalmente, si los programas a desarrollar son muy simples, se puede utilizar un recurso más barato. Sin embargo, si se tienen que desarrollar programas complejos es difícil justificar el ahorro en hardware cuando se compra un sistema de desarrollo, pues los costos de programación resultantes superarán en mucho tales ahorros.

Para una persona aficionada a los ordenadores, un microordenador individual ofrecerá recursos suficientes, aunque mínimos. Un buen software de desarrollo está aún por llegar para la mayoría de los ordenadores individuales. El usuario tendrá que evaluar su sistema en función de los comentarios presentados en este capítulo.

Analicemos más detalladamente el recurso más indispensable: el ensamblador.

EL ENSAMBLADOR

Hemos utilizado el lenguaje en nivel ensamblador a lo largo de este libro sin presentar la sintaxis o definiciones formales del lenguaje en nivel ensamblador y por ello se van a presentar estas definiciones. Un ensamblador se concibe para proporcionar una representación simbólica cómoda del programa del usuario, proporcionando, al mismo tiempo, un medio sencillo de convertir estos nemónicos en su representación binaria.

Campos del ensamblador

Hemos visto que cuando se escribe un programa para el ensamblador, se utilizan los siguientes campos:

El campo de etiqueta, opcional, que puede contener una dirección simbólica para la instrucción que sigue a continuación.

El campo de instrucción, que incluye el código de operación y los operandos (se puede separar un campo operando independiente).

DIRECCIÓN	INSTRUCCIÓN HEXADECIMAL			ETIQUETA	OPCODE (CÓDIGO DE OPERACIÓN SIMBÓLICO)	OPERANDO	COMENTARIOS
	1	2	3				

Figura 10-5 Formato de programación para microprocesador.

El campo de comentario, más a la derecha, que es opcional y está concebido para explicar el programa.

Una vez que el programa ha sido entregado al ensamblador, este último produce un *listado* del mismo. Cuando se genera un listado, el ensamblador proporciona tres campos adicionales, habitualmente a la izquierda de la página. En la figura 10-6 se muestra un ejemplo. En la parte más a la izquier-

```
LINE # LOC     CODE        LINE

0057   0342  A9 00                LDA #$00
0058   0344  8D 0B A0             STA ACR1      ;TURN BOTH TIMERS OFF
0059   0347  8D 0B AC             STA ACR2
0060   034A  A2 20                LDX #OFFDEL   ;GET TONES-OFF DELAY CONSTANT
0061   034C  20 55 03    OFF      JSR DELAY     ;DELAY WHILE TONE IS OFF
0062   034F  CA                   DEX
0063   0350  D0 FA                BNE OFF
0064   0352  4C 02 03             JMP DIGIT     ;GO BACK FOR NEXT DIGIT OF PHONE NUMBER
0065   0355              ;
0066   0355              ;THIS IS A SIMPLE DELAY ROUTINE FOR THE TONE ON AND OFF PERI
0067   0355              ;
0068   0355  A9 FF       DELAY    LDA #DELCON   ;GET DELAY CONSTANT
0069   0357  38          WAIT     SEC           ;DELAY FOR THAT LONG
0070   0358  E9 01                SBC #$01
0071   035A  D0 FB                BNE WAIT
0072   035C  60                   RTS
0073   035D              ;
0074   035D              ;THIS IS A TABLE OF THE CONSTANTS FOR THE TONE FREQUENCIES
0075   035D              ;FOR EACH TELEPHONE DIGIT.  THE CONSTANTS ARE TWO BYTES
0076   035D              ;LONG, LOW BYTE FIRST.
0077   035D              ;
0078   035D  13          TABLE  .BYTE $13,$02,$76,$01 ;TWO TONES FOR '0'
0078   035E  02
0078   035F  76
0078   0360  01
0079   0361  CD                 .BYTE $CD,$02,$9E,$01 ;TWO TONES FOR '1'
0079   0362  02
0079   0363  9E
0079   0364  01
0080   0365  CD                 .BYTE $CD,$02,$76,$01 ;      '2'
0080   0366  02
0080   0367  76
0080   0368  01
0081   0369  CD                 .BYTE $CD,$02,$53,$01 ;         '3'
0081   036A  02
0081   036B  53
0081   036C  01
0082   036D  89                 .BYTE $89,$02,$9E,$01 ;      '4'
0082   036E  02
0082   036E  9E
0082   0370  01
0083   0371  89                 .BYTE $89,$02,$76,$01 ;      '5'
0083   0372  02
0083   0373  76
0083   0374  01
0084   0375  89                 .BYTE $89,$02,$53,$01 ;      '6'
0084   0376  02
0084   0377  53
0084   0378  01
0085   0379  4B                 .BYTE $4B,$02,$9E,$01 ;      '7'
0085   037A  02
0085   037B  9E
0085   037C  01
0086   037D  4B                 .BYTE $4B,$02,$76,$01 ;    '8'
0086   037E  02

LINE # LOC     CODE        LINE

0086   037E  76
0086   0380  01
0087   0381  4B                 .BYTE $4B,$02,$53,$01 ;       '9'
0087   0382  02
0087   0383  53
0087   0384  01
0088   0385                     .END

SYMBOL TABLE

SYMBOL    VALUE

ACR1      A00B   ACR2    AC0B   DELAY   03[illegible]   DELCON  00FF
DIGIT     0302   NOEND   030A   NUMPTR  00[illegible]   OFF     034C
OFFDEL    0020   ON      033C   ONDEL   0040   PHONE   0300
T1CH      A005   T1LH    A007   T1LL    A004   T2CH    AC05
T2LH      AC07   T2LL    AC04   TABLE   035D   WAIT    0357

END OF ASSEMBLY
```

Figura 10-6 Ejemplo de salida del ensamblador.

da está el número de línea. Cada línea escrita por el programador se asigna a un número de línea simbólica.

El campo que sigue a la derecha es el campo de dirección real, que muestra el valor hexadecimal del contador de programa que apuntará a esta instrucción.

El siguiente campo a la derecha es la representación hexadecimal de la instrucción.

Éste muestra uno de los posibles empleos de un ensamblador. Incluso si se preparan programas para un microordenador de una sola tarjeta que acepta solamente el hexadecimal, se deben escribir siempre los programas en lenguaje en nivel ensamblador, esforzándose en tener acceso a un sistema provisto de ensamblador. Se pueden ejecutar entonces los programas en el sistema, utilizando el ensamblador. Éste generará automáticamente los códigos hexadecimales correctos, que se pueden escribir simplemente en nuestro sistema. Esto muestra, en un ejemplo sencillo, el interés de los recursos software adicionales.

Tablas

Cuando el ensamblador convierte el programa simbólico en su representación binaria, realiza dos tareas esenciales:

1. Traduce las instrucciones nemónicas en su codificación binaria.
2. Traduce los símbolos utilizados para las constantes y las direcciones en su representación binaria.

Para facilitar la depuración del programa, el ensamblador imprime al final del listado la correspondencia entre cada símbolo utilizado y su valor hexadecimal equivalente. Es lo que se llama tabla de símbolos.

Algunas tablas de símbolos no solamente listarán el valor y su símbolo, sino también los números de línea en donde aparece el símbolo. Se trata de un recurso adicional.

Mensajes de error

Durante el proceso de ensamblado, el ensamblador detectará errores de sintaxis y los lista como parte del listado final. Los diagnósticos típicos incluyen: símbolos no definidos, etiquetas ya definidas, códigos de operación ilegales, direcciones ilegales y modos de direccionamiento prohibidos. Muchos otros diagnósticos de detalle son naturalmente deseables y suelen ser proporcionados. Varían con cada ensamblador.

El lenguaje ensamblador

Los códigos de operación han sido ya definidos. Definiremos ahora los símbolos, constantes y operadores que pueden ser utilizados como parte de la sintaxis del ensamblador.

Símbolos

Se utilizan para representar valores numéricos, datos o direcciones. Tradicionalmente, los símbolos pueden incluir 6 caracteres, de los cuales el primero debe ser alfabético. Existe una restricción más: los 56 códigos de operación utilizados por el 6502 y los nombres de los registros, es decir, A, X, Y, S, P pueden no ser utilizados como símbolos.

Asignación de un valor a un símbolo

Las etiquetas son símbolos especiales cuyos valores no necesitan ser definidos por el programador. Ellos corresponden automáticamente al número de línea en donde aparecen. Sin embargo, los otros símbolos utilizados para constantes o direcciones de memoria se deben definir por el programador antes de su empleo. El signo igual se utiliza para este propósito o bien un "directivo" (pseudoinstrucción) especial. Es una instrucción para el ensamblador que no será convertida en una sentencia ejecutable, se denomina *directivo ensamblador.*

Por ejemplo, la constante ALPHA podría definirse como:

ALPHA = $A000

Ello asigna el valor "A000" hexadecimal a la variable ALPHA. Los directivos del ensamblador se examinarán en una sección posterior.

Constantes o literales

Las constantes se expresan tradicionalmente en decimal, hexadecimal, octal, o binario. Excepto en el caso de un número decimal, se utiliza un prefijo para diferenciar entre una constante y la base utilizada para representar un número. Para cargar 18 en el acumulador se escribirá simplemente:

LDA # 18 (en donde # significa un literal).

Un número hexadecimal irá precedido por el símbolo $.
Un símbolo octal será precedido por el símbolo @
Un símbolo binario será precedido por %.

Por ejemplo, para cargar el valor "11111111" en el acumulador, escribiremos:

LDA #%11111111

Los caracteres literales ASCII se pueden también utilizar en un campo literal. En los ensambladores antiguos, era tradicional encerrar el símbolo ASCII entre comillas. En los ensambladores más modernos, para tener que introducir menos caracteres, el tipo alfanumérico se indica por un simple apóstrofo que precede al símbolo.

Por ejemplo, para cargar el símbolo "S" en el acumulador (en ASCII) se escribirá:

LDA #'S

Para poder cargar el símbolo de comillas propiamente dicho, la notación es:

LDA #'''

Ejercicio 10.1: *¿Cargan las dos instrucciones siguientes el mismo valor en el acumulador:* LDA#'5 *y* LDA#$5*?*

Operadores

Para facilitar todavía más la escritura posterior de programas simbólicos, los ensambladores permiten el empleo de operadores. Como mínimo, deben permitir "más" y "menos", de modo que se puede especificar, por ejemplo:

LDA ADR1, y
LDX ADR1 + 1

Es importante comprender que la expresión ADR1 + 1 será calculada por el ensamblador, para determinar cuál es la dirección verdadera que se debe insertar como equivalente binario. Será calculada *en el momento del ensamblado* y no en el instante de ejecución del programa.

Además pueden estar disponibles muchos operadores tales como multiplicación y división, lo que es cómodo para acceder a tablas en memoria. También se puede disponer de operadores más especializados, tales como, "mayor que" y "menor que", los cuales truncan un valor de 2 bytes en su byte alto y bajo respectivamente.

Naturalmente, el resultado de una expresión debe ser un valor positivo. Los números negativos no suelen utilizarse y se deben expresar en un formato hexadecimal.

Finalmente, de manera usual se utiliza un símbolo especial para repre-

sentar el valor corriente de la dirección de la línea :*. Este símbolo debe ser interpretado como "posición corriente" (valor de PC).

Ejercicio 10.2: *¿Cuál es la diferencia entre las siguientes instrucciones?*

```
LDA %10101010
LDA #%10101010
```

Ejercicio 10.3: *¿Cuál es el efecto de la instrucción* BMI*−2?

Directivos de ensamblador

Los directivos (pseudoinstrucciones) son órdenes especiales dadas por el programador al ensamblador. Algunas de estas órdenes dan lugar al almacenamiento de valores en símbolos o en la memoria. Otros se utilizan para controlar la ejecución o modo de impresión del ensamblador.

Para dar un ejemplo concreto pasemos revista a los nueve directivos de ensamblador disponibles en el sistema de desarrollo Rockwell ("Sistema 65"). Son: =, .BYT, .WOR, .GBY, .PAGE, .SKIP, .OPT, .FILE y .END.

Directivo de equivalencia

Se utiliza un signo igual para asignar un valor numérico a un símbolo. Por ejemplo:

```
BASE = $1111
* = $1234
```

El efecto del primer directivo es asignar el valor 1111 hexadecimal a BASE.

El efecto de la segunda instrucción es forzar la dirección de la línea al valor hexadecimal "1234". Dicho de otro modo, la siguiente instrucción ejecutable que se encuentre se almacenará en la posición de memoria 1234.

Ejercicio 10.4: *Escribir un directivo que haga residir el programa a partir de la posición de memoria 0.*

Directivos para inicializar la memoria

Para este fin están disponibles tres directivos: .BYT, .WOR y .GBY. .BYT asignará los caracteres o valores que siguen en bytes de memoria consecutivos.

Ejemplo: RESERV .BYT 'SYBEX.'

Ello tendrá por resultado almacenar las letras "SYBEX" en posiciones de memoria consecutivas.

.WOR se utiliza para almacenar direcciones de 2 bytes en memoria, con el byte bajo en primer lugar.

Ejemplo: .WOR $1234, $2345

.GBY es idéntico a .WOR salvo que almacenará un valor de 16 bits, con el byte alto en primer lugar. Suele utilizarse para datos de 16 bits más que para direcciones de 16 bits.

Los tres directivos siguientes se utilizan para controlar las entradas/salidas.

Directivos de entrada/salida

Los directivos de entrada/salida son: .PAGE, .SKIP y .OPT.

.PAGE hace que el ensamblador termine la página, es decir, que se desplace a la parte superior de la siguiente página. Además, un título se puede especificar por página. Por ejemplo: .PAGE "título de página".

.SKIP sirve para insertar líneas en blanco en el listado o se puede especificar el número de líneas a saltar. Por ejemplo: SKIP 3.

.OPT especifica cuatro opciones: listar, generar, errores y símbolo. *List* generará una lista. *Generate* se utiliza para imprimir el código objeto de cadenas con el directivo .BYT; *Error* especifica si el diagnóstico de error se debe imprimir. *Symbol* especifica si se debe listar la tabla de símbolos.

Los dos últimos directivos controlan el formato del listado ensamblador.

Directivos .FILE y .END

En el desarrollo de un programa grande se escribirán y depurarán por separado varias partes del programa. En cierto momento será necesario ensamblarlos a partir de estos ficheros. La última sentencia del primer fichero incluirá entonces el directivo .FILE NAME/1, en donde 1 es el número de la unidad de disco y NAME es el nombre del siguiente fichero, que, a su vez, puede estar enlazado a más ficheros. Al final del último fichero estará el directivo: .END NAME/1, que apunta hacia el primero.

Finalmente, existe un medio para insertar comentarios suplementarios en el listado: ";".

";" se puede utilizar para introducir comentarios, a voluntad, en el interior de una línea, en lugar de introducir una instrucción. Es un recurso importante si se desea que los programas estén documentados correctamente.

MACROS

No hay actualmente posibilidades de macros en la mayoría de los ensambladores que existen para el 6502. Sin embargo, definiremos lo que es un macro y cuáles son sus ventajas. Se puede esperar que habrá pronto posibilidades de macros en la mayoría de los ensambladores 6502.

Un macro no es más que un nombre asignado a un grupo de instrucciones. Es, esencialmente, una comodidad para el programador. Por ejemplo, si un grupo de cinco instrucciones se utiliza varias veces en un programa, se puede definir un macro en vez de tener que escribir cada vez estas cinco instrucciones. Por ejemplo, se puede escribir:

```
SAVREG MACRO PHA
             TXA
             PHA
             TYA
             PHA
       ENDM
```

En lo sucesivo se podrá escribir el nombre SAVREG en lugar de las instrucciones anteriores.

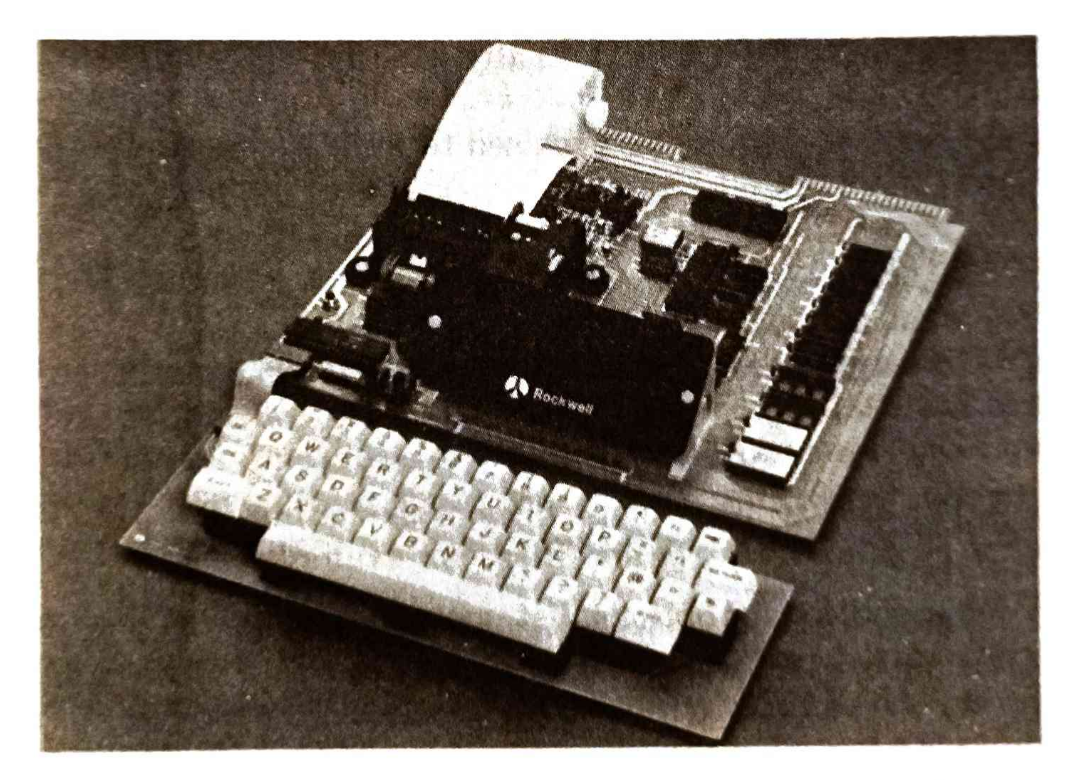


Figura 10-7 El AIM 65 es una tarjeta con una miniimpresora y un teclado completo.



Figura 10-8 Ohio Scientific es un microordenador personal.

Cada vez que se escribe SAVREG, las cinco líneas correspondientes se sustituyen en vez del nombre. Un ensamblador que posee la posibilidad de los macros se llama macroensamblador. Cuando el macroensamblador encuentra SAVREG, se efectuará meramente una sustitución física de las líneas equivalentes.

¿Macro o subrutina?

En este momento puede parecer que un macro funciona de modo análogo a una subrutina, pero no es así. Cuando el ensamblador se utiliza para producir el código objeto, cada vez que se encuentre un nombre de macro, será sustituido por las instrucciones reales que representa. En el momento de la ejecución, el grupo de instrucciones aparecerá tantas veces como aparezca el nombre de macro.

Por el contrario, la subrutina se define solamente una vez y, por ello, puede ser utilizada repetidamente: el programa saltará a la dirección de la subrutina. Un macro es un recurso que se llama en el *momento del ensamblado.* Una subrutina es un recurso aplicable en el *momento de la ejecución.* Su funcionamiento es bastante diferente.

Parámetros de los macros

Cada macro puede poseer un cierto número de parámetros. Por ejemplo, consideremos el siguiente macro:

```
SWAP   MACRO   M, N, T
       LDA     M
       STA     T
       LDA     N
       STA     M
       LDA     T
       STA     N
       ENDM
```

Este macro dará lugar a un intercambio de los contenidos de las posiciones de memoria M y N. Un intercambio entre dos registros, o dos posiciones de memoria, es una operación que no proporciona el 6502. Un macro se puede utilizar para realizarla. "T", en este caso, es simplemente el nombre de una posición de almacenamiento temporal necesario para el programa. Por ejemplo, intercambiemos los contenidos de las posiciones de memoria ALPHA y BETA. La instrucción que hace este intercambio se muestra a continuación:

```
SWAP   ALPHA, BETA, TEMP
```

En esta instrucción, TEMP es el nombre de una posición de memoria temporal, que sabemos está libre y que se puede utilizar para el macro. El resultado del desarrollo del macro es:

```
LDA   ALPHA
STA   TEMP
LDA   BETA
STA   ALPHA
LDA   TEMP
STA   BETA
```

La importancia de un macro será ahora evidente: es una gran comodidad para el programador poder utilizar pseudoinstrucciones que se han definido con los macros. De este modo, el juego de instrucciones aparente del 6502 se puede ampliar a voluntad. Lamentablemente, es preciso tener presente que todo macro se desarrollará en tantas instrucciones como había en su definición. Un macro será, pues, ejecutado más lentamente que cualquier instrucción sencilla. Debido a su comodidad para el desarrollo de cualquier programa largo, el macro es un recurso muy deseable para tales aplicaciones.

Posibilidades suplementarias de los macros

Se pueden añadir otras muchas posibilidades a los macros simples, tales como otras pseudoinstrucciones y recursos sintácticos. Por ejemplo, los macros pueden ser *anidados,* es decir, una llamada a un macro puede aparecer en el interior de una definición macro. Utilizando esta posibilidad, un macro se puede modificar a sí mismo con una definición anidada. Una primera llamada producirá un desarrollo modificado del mismo macro.

ENSAMBLADO CONDICIONAL

El ensamblado condicional es otro recurso del ensamblador, del que hasta este momento carecían la mayoría de los ensambladores del 6502. Un ensamblador condicional permite al programador utilizar instrucciones especiales "IF", seguidas por una expresión, después (opcionalmente) "ELSE" y finalmente "ENDIF". Cuando la expresión que sigue al IF es verdadera, entonces las instrucciones entre el IF y el ELSE o el IF y el ENDIF (si no hay ELSE) será ensambladas. En el caso de que se utilice IF seguida de ELSE, uno u otro de los dos bloques de instrucciones será ensamblado, dependiendo del valor de la expresión que se está comprobando.

Con la posibilidad del ensamblado condicional, el programador puede preparar programas para diferentes casos y después ensamblar condicionalmente los segmentos de los códigos requeridos para una aplicación concreta. Por ejemplo, un usuario industrial puede concebir programas capaces de ocuparse de cualquier número de luces de tráfico en una intersección para un conjunto de algoritmos de control. Recibirá después las especificaciones del ingeniero de tráfico local, quien le va a indicar cuántas luces de tráfico deberán existir y cuántos algoritmos se deben utilizar. El programador sólo tendrá que establecer parámetros en su programa y efectuar el ensamblado condicional. El ensamblado condicional producirá un programa adaptado al cliente que no retendrá más que aquellas rutinas que sean necesarias para la solución del problema.

El ensamblado condicional es, pues, de interés particular para la generación de programas industriales en un contexto en donde existen muchas opciones y en donde el programador desea ensamblar partes de programas rápida y automáticamente en respuesta a parámetros externos.

RESUMEN

Este capítulo ha presentado una explicación de las técnicas y de las

herramientas hardware y software requeridas para desarrollar un programa, así como las diferentes alternativas y soluciones de compromiso.

Se extienden, en el nivel hardware, desde el microordenador de una sola tarjeta al sistema de desarrollo completo; en el nivel de software, desde la codificación binaria a la programación de alto nivel. Tendrá que seleccionar entre estas herramientas y técnicas de acuerdo con sus objetivos y presupuesto.

11 Conclusión

Hemos tratado hasta ahora todos los aspectos importantes de la programación, incluyendo las definiciones y conceptos básicos, las manipulaciones internas de los registros del 6502, la gestión de los dispositivos de entrada/salida y las características de las ayudas al desarrollo del software. ¿Cuál es la etapa siguiente? Se pueden considerar dos aspectos, el primero relacionado con el desarrollo de la tecnología, el segundo concerniente al desarrollo de sus propios conocimientos y aptitudes. Examinemos estos dos puntos.

DESARROLLO TECNOLÓGICO

El progreso de la integración en tecnología MOS hace posible la realización de pastillas cada vez más complejas. El coste de la realización de la función procesador propiamente dicha decrece constantemente. El resultado es que muchas de las pastillas de entrada/salida, así como las pastillas controladoras de periféricos, utilizadas en un sistema, incorporan ahora un procesador sencillo. Ello significa que las pastillas LSI utilizadas actualmente en un sistema se han hecho *programables*. Aparece, pues, un dilema interesante a nivel conceptual. Para simplificar la tarea de concepción de software, y reducir también el número de componentes, las nuevas pastillas de E/S incorporan actualmente posibilidades programables complejas: numerosos algoritmos programados están integrados en la pastilla. Sin embargo, resulta de ello que el desarrollo de programas se complica por el hecho de que todas estas pastillas de entrada/salida son muy diferentes y necesitan ser estudiadas detalladamente por el programador. *Programar el sistema no es solamente*



Figura 11-1 El PET es una unidad integrada.



Figura 11-2 El Apple II utiliza una pantalla de TV convencional.

programar el microprocesador, sino también programar las otras diferentes pastillas conectadas al mismo. El tiempo de aprendizaje de cada una de estas pastillas puede ser considerable.

Naturalmente, el dilema no es más que aparente. Si estas pastillas no estaban disponibles, la complejidad del interface a realizar, así como los programas correspondientes, serán todavía mayores. La complejidad nueva que se introduce es que se ha de programar más como procesador, y aprender las diversas características de las diferentes pastillas en un sistema para utilizarlas eficazmente. Sin embargo, se espera que las técnicas y los conceptos presentados en este libro hagan que ello sea una tarea razonablemente fácil.

LA ETAPA SIGUIENTE

Ha aprendido ahora las técnicas básicas requeridas para poder programar, en papel, aplicaciones sencillas. Era el objetivo de este libro. La siguiente etapa es la práctica real. No tiene sustitución posible. Es imposible aprender la programación únicamente sobre el papel y se requiere experiencia. Ahora debe estar en condiciones de comenzar a escribir sus propios programas. Confiamos en que este camino le haya resultado agradable.

APÉNDICE A

TABLA DE CONVERSIÓN HEXADECIMAL

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00	000
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

5		4		3		2		1		0	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6.291,456	6	393.216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728.640	F	983.040	F	61,440	F	3,840	F	240	F	15

APÉNDICE B

INSTRUCCIONES DEL 6502 POR ORDEN ALFABÉTICO

ADC	Suma con acarreo.
AND	Operación AND lógica.
ASL	Desplazamiento aritmético a la izquierda.
BCC	Bifurcación si el bit de acarreo es 0.
BCS	Bifurcación si el bit de acarreo es 1.
BEQ	Bifurcación si el resultado es igual a 0.
BIT	Comprobación de bits.
BMI	Bifurcación si el resultado es negativo.
BNE	Bifurcación si el resultado no es igual a 0.
BPL	Bifurcación si el resultado es positivo.
BRK	Ruptura.
BVC	Bifurcación si el bit de desbordamiento es 0.
BVS	Bifurcación si el bit de desbordamiento es 1.
CLC	Pone a cero el bit de acarreo.
CLD	Pone a cero el bit de modo decimal.
CLI	Pone a cero el indicador I de interrupciones.
CLV	Pone a cero el bit de desbordamiento.
CMP	Comparar con acumulador.
CPX	Comparar con X.
CPY	Comparar con Y.
DEC	Decrementar memoria.
DEX	Decrementar X.
DEY	Decrementar Y.
EOR	OR exclusiva.
INC	Incrementar memoria.
INX	Incrementar X.
INY	Incrementar Y.
JMP	Salto.
JSR	Salto a subrutina.
LDA	Cargar acumulador.
LDX	Cargar X.
LDY	Cargar Y.

LSR	Desplazamiento lógico a la derecha.
NOP	No operación.
ORA	OR lógica.
PHA	Introducir en pila A.
PHP	Introducir en pila el estado del procesador.
PLA	Extraer acumulador.
PLP	Extraer el estado del procesador desde la pila.
ROL	Rotación a la izquierda.
ROR	Rotación a la derecha.
RTI	Retorno desde una interrupción.
RTS	Retorno desde una subrutina.
SBC	Resta con acarreo.
SEC	Pone a 1 el indicador de acarreo C.
SED	Pone a 1 el indicador decimal D.
SEI	Pone a 1 el indicador de inhibición de interrupciones I.
STA	Almacenar acumulador.
STX	Almacenar X.
STY	Almacenar Y.
TAX	Transferir A a X.
TAY	Transferir A a Y.
TSX	Transferir SP a X.
TXA	Transferir X a A.
TXS	Transferir X a SP.
TYA	Transferir Y a A.

APÉNDICE C

LISTA BINARIA DE LAS INSTRUCCIONES DEL 6502

Ver capítulo 4 para definición del campo "bb".

ADC	011bbb01	JSR	00100000
AND	001bbb01	LDA	101bbb01
ASL	000bbb10	LDX	101bbb10
BCC	10010000	LDY	101bbb00
BCS	10110000	LSR	010bbb10
BEQ	11110000	NOP	01bbb110
BIT	0010b100	ORA	000bbb01
BMI	00110000	PHA	01001000
BNE	11010000	PHP	00001000
BPL	00010000	PLA	01101000
BRK	00000000	PLP	00101000
BVC	01010000	ROL	001bbb10
BVS	01110000	ROR	011bbb10
CLC	00011000	RTI	01000000
CLD	11011000	RTS	01100000
CLI	01011000	SBC	111bbb01
CLV	10111000	SEC	00111000
CMP	110bbb01	SED	11111000
CPX	1110bb00	SEI	01111000
CPY	1100bb00	STA	100bbb01
DEC	110bb110	STX	100bb110
DEX	11001010	STY	100bb100
DEY	10001000	TAX	10101010
EOR	110bbb01	TAY	10101000
INC	111bb110	TSX	10111010
INX	11101000	TXA	10001010
INY	11001000	TXS	10011010
JMP	01b01100	TYA	10011000

APÉNDICE D

JUEGO DE INSTRUCCIONES DEL 6502: HEXADECIMAL Y DURACIÓN

n = número de ciclos   # = número de bytes

		IMPLICADO			ACUMULADOR			ABSOLUTO			PAGINA 0			INMEDIATO			ABS X			ABS Y		
MNEMONICO		OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#
ADC	(1)							6D	4	3	65	3	2	69	2	2	7D	4	3	79	4	3
AND	(1)							2D	4	3	25	3	2	29	2	2	3D	4	3	39	4	3
ASL					OA	2	1	OE	6	3	06	5	2				1E	7	3			
BCC	(2)																					
BCS	(2)																					
BEQ	(2)																					
BIT								2C	4	3	24	3	2									
BMI	(2)																					
BNE	(2)																					
BPL	(2)																					
BRK		00	7	1																		
BVC	(2)																					
BVS	(2)																					
CLC		18	2	1																		
CLD		D8	2	1																		
CLI		58	2	1																		
CLV		B8	2	1																		
CMP								CD	4	3	C5	3	2	C9	2	2	DD	4	3	D9	4	3
CPX								EC	4	3	E4	3	2	EO	2	2						
CPY								CC	4	3	C4	3	2	CO	2	2						
DEC								CE	6	3	C6	5	2				DE	7	3			
DEX		CA	2	1																		
DEY		88	2	1																		
EOR	(1)							4D	4	3	45	3	2	49	2	2	5D	4	3	59	4	3
INC								EE	6	3	E6	5	2				FE	7	3			
INX		E8	2	1																		
INY		C8	2	1																		
JMP								4C	3	3												
JSR								20	6	3												
LDA	(1)							AD	4	3	A5	3	2	A9	2	2	BD	4	3	B9	4	3
LDX	(1)							AE	4	3	A6	3	2	A2	2	2				BE	4	3
LDY	(1)							AC	4	3	A4	3	2	A0	2	2	BC	4	3			
LSR					4A	2	1	4E	6	3	46	5	2				5E	7	3			
NOP		EA	2	1																		
ORA								0D	4	3	05	3	2	09	2	2	1D	4	3	19	4	3
PHA		48	3	1																		
PHP		08	3	1																		
PLA		68	4	1																		
PLP		28	4	1																		
ROL					2A	2	1	2E	6	3	26	5	2				3E	7	3			
ROR					6A	2	1	6E	6	3	66	5	2				7E	7	3			
RTI		40	6	1																		
RTS		60	6	1																		
SBC	(1)							ED	4	3	E5	3	2	E9	2	2	FD	4	3	F9	4	3
SEC		38	2	1																		
SED		F8	2	1																		
SEI		78	2	1																		
STA								8D	4	3	85	2					9D	5	3	99	5	3
STX								8E	4	3	86	2										
STY								8C	4	3	84	2										
TAX		AA	2	1																		
TAY		A8	2	1																		
TSX		BA	2	1																		
TXA		8A	2	1																		
TXS		9A	2	1																		
TYA		98	2	1																		

(1) Sumar 1 a n si se cruza el límite de la página.

(IND X)			(IND)Y			PAGINA 0, X			RELATIVO			INDIRECTO			PAGINA 0, Y			CÓDIGOS DE ESTADOS DEL PROCESADOR								
OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	N	V		B	D	I	Z	C	MNEMONICO
61	6	2	71	5	2	75	4	2										•	•					•	•	ADC
21	6	2	31	5	2	35	4	2										•						•		AND
						16	6	2										•						•	•	ASL
									90	2	2															BCC
									B0	2	2															BCS
									F0	2	2															BEQ
																		M_7	M_6					•		BIT
									30	2	2															BMI
									D0	2	2															BNE
									10	2	2															BPL
																					1		1			BRK
									50	2	2															BVC
									70	2	2															BVS
																									0	CLC
																						0				CLD
																							0			CLI
																			0							CLV
C1	6	2	D1	5	2	D5	4	2										•						•	•	CMP
																		•						•	•	CPX
																		•						•	•	CPY
						D6	6	2										•						•		DEC
																		•						•		DEX
																		•						•		DEY
41	6	2	51	5	2	55	4	2										•						•		EOR
						F6	6	2										•						•		INC
																		•						•		INX
																		•						•		INY
												6C	5	3												JMP
																										JSR
A1	6	2	B1	5	2	B5	4	2										•						•		LDA
															B6	4	2	•						•		LDX
						B4	4	2										•						•		LDY
						56	6	2										0						•	•	LSR
																										NOP
01	6	2	11	5	2	15	4	2										•						•		ORA
																										PHA
																										PHP
																		•						•		PLA
																		•	•	•	•	•	•	•	•	PLP
						36	6	2										•						•	•	ROL
						76	6	2										•						•	•	ROR
																		•	•	•	•	•	•	•	•	RTI
																										RTS
E1	6	2	F1	5	2	F5	4	2										•	•					•	•	SBC
																									1	SEC
																						1				SED
																							1			SEI
81	6	2	91	6	2	95	4	2																		STA
															96	4	2									STX
						94	4	2																		STY
																		•						•		TAX
																		•						•		TAY
																		•						•		TSX
																		•						•		TXA
																										TXS
																		•						•		TYA

(2) Sumar 2 a n si se bifurca dentro de la página. Sumar 3 a n si se bifurca a otra página.

APÉNDICE E

TABLA DE CONVERSIÓN ASCII

Código	Carácter	Código	Carácter	Código	Carácter	Código	Carácter
00	NUL	20[1]		40	@	60[5]	`
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27[2]	'	47	G	67	g
08	BS	28	(	48	H	68	h
09	TAB	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C[3]	,	4C	L	6C	l
0D	CR	2D	—	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	\	7C	|
1D	GS	3D	=	5D	]	7D[6]	}
1E	RS	3E	>	5E	↑	7E	~
1F	US	3F	?	5F[4]	←	7F[7]	BORRADO

[1] espacio
[2] notación
[3] coma
[4] o subrayado
[5] acento
[6] o MODO ALTO
[7] o DEL

APÉNDICE F

TABLAS DE BIFURCACIÓN RELATIVAS

BIFURCACIÓN RELATIVA DIRECTA

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

BIFURCACIÓN RELATIVA INDIRECTA

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

APÉNDICE G

LISTA POR CÓDIGOS DE OPERACIÓN EN HEXADECIMAL

LSD / MSD	0	1	2	3	4	5	6	7
0	BRK	ORA-I, X				ORA Ø P	ASL Ø P	
1	BPL	ORA-I, Y				ORA-Ø P, X	ASL Ø P, X	
2	JSR	AND-I, X			BIT-Ø-P	AND Ø P	ROL Ø P	
3	BMI	AND-I, Y				AND Ø P, X	ROL Ø P, X	
4	RTI	EOR-I, X				EOR Ø P	LSR Ø P	
5	BVC	EOR-I, Y				EOR-Ø-P, X	LSR Ø P, X	
6	RTS	ADC-I, X				ADC Ø P	ROR Ø P	
7	BVS	ADC-I, Y				ADC-Ø P, X		
8		STA-I, X			STY-Ø-P	STA-Ø-P	STX Ø P	
9	BCC	STA-I, Y			STY-Ø-P, X	STA-Ø P, X	STX Ø P, Y	
A	LDY-IMM	LDA-I, X	LDX-IMM		LDY-Ø-P	LDA-Ø-P	LDX Ø P	
B	BCS	LDA-I, Y			LDY-Ø-P, X	LDA-Ø P, X	LDX-Ø-P, Y	
C	CPY-IMM	CMP-I, X			CPY-Ø-P	CMP Ø P	DEC Ø P	
D	BNE	CMP-I, Y				CMP-Ø P, X	DEC-Ø-P, X	
E	CPX-IMM	SBC-I, X			CPX-Ø-P	SBC-Ø-P	INC Ø P	
F	BEQ	SBC-I, Y				SBC-Ø P, X	INC-Ø-P, X	

8	9	A	B	C	D	E	F	LSD / MSD
PHP	ORA-IMM	ASL-A			ORA	ASL		0
CLC	ORA, Y				ORA, X	ASL, X		1
PLP	AND-IMM	ROL A		BIT	AND	ROL		2
SEC	AND, Y				AND, X	ROL, X		3
PHA	EOR-IMM	LSR A		JMP	EOR	LSR		4
CLI	EOR, Y				EOR, X	LSR, X		5
PLA	ADC-IMM	ROR A		JMP I	ADC	ROR		6
SEI	ADC, Y				ADC, X			7
DEY		TXA		STY	STA	STX		8
TYA	STA, Y	TXS			STA, X			9
TAY	LDA-IMM	TAX		LDY	LDA	LDX		A
CLV	LDA, Y	TSX		LDY, X	LDA, X	LDX, Y		B
INY	CMP-IMM	DEX		CPY	CMP	DEC		C
CLD	CMP, Y				CMP, X	DEC, X		D
INX	SBC-IMM	NOP		CPX	SBC	INC		E
SED	SBC, Y				SBC, X	INC, X		F

I = indirecto

ØP = página cero

APÉNDICE H

CONVERSIÓN DECIMAL A BCD

DECIMAL	BCD	DEC	BCD	DEC	BCD
0	0000	10	00010000	90	10010000
1	0001	11	00010001	91	10010001
2	0010	12	00010010	92	10010010
3	0011	13	00010011	93	10010011
4	0100	14	00010100	94	10010100
5	0101	15	00010101	95	10010101
6	0110	16	00010110	96	10010110
7	0111	17	00010111	97	10010111
8	1000	18	00011000	98	10011000
9	1001	19	00011001	99	10011001

APÉNDICE I

SOLUCIONES DE LOS EJERCICIOS

CAPÍTULO 1

1.1: 252

1.2: 100000001

1.3: $19 \div 2 = 9$ resto $1 \rightarrow 1$
$9 \div 2 = 4$ resto $1 \rightarrow 1$
$4 \div 2 = 2$ resto $0 \rightarrow 0$
$2 \div 2 = 1$ resto $0 \rightarrow 0$
$1 \div 2 = 0$ resto $1 \rightarrow 1$
Solución: 10011

$$\begin{array}{r} 1 \times 1 = 1 \\ 1 \times 2 = 2 \\ 0 \times 4 = 0 \\ 0 \times 8 = 0 \\ + \; 1 \times 16 = 16 \\ \hline \text{Solución:} \quad 19 \end{array}$$

1.4:
$$\begin{array}{r} 0101 = 5 \\ + \; 1010 = 10 \\ \hline 1111 = 15 \end{array}$$

$$\begin{array}{r} 1 \times 1 = 1 \\ 1 \times 2 = 2 \\ 1 \times 4 = 4 \\ + \; 1 \times 8 = 8 \\ \hline \text{Solución:} \quad 15 \end{array}$$

1.5: 1111
+ 0001
---
(1) 0000
Solución: No, el resultado no tiene 4 bits.

1.6: +5 = 00000101
−5 = 10000101

1.7: +6 = 00000110
−6 = 11111001

1.8: +127 = 01111111

1.9: +128 = 10000000
01111111 (complemento a uno)
+ 1

−128 = 10000000 (complemento a dos)

1.10: El mayor: −128
El menor: +127

1.11: +20 = 00010100
11101011 (complemento a uno)
+ 1

−20 = 11101100 (complemento a dos)
00010011 (complemento a dos)
+ 1
---
20 = 00010100
Respuesta: Sí.

1.12: 10111111
+ 11000001

10000000

V: 0 C: 1
☒ CORRECTO

```
  11111010
+ 11111001
----------
  11110011
```

V:0 C:1
☒ CORRECTO

```
  00010000
+ 01000000
----------
  01010000
```

V:0 C:0
☒ CORRECTO

```
  01111110
+ 00101010
----------
  10101000
```

V:1 C:0
☒ ERROR

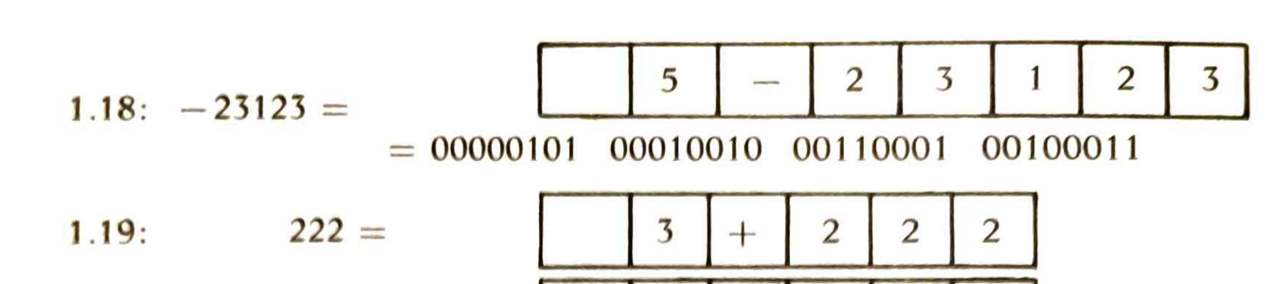
1.13: No; no es posible generar un desbordamiento cuando se suman un número positivo y un número negativo, a causa de que tienden a cancelarse mutuamente; el resultado está, pues, siempre dentro del margen de 1 byte.

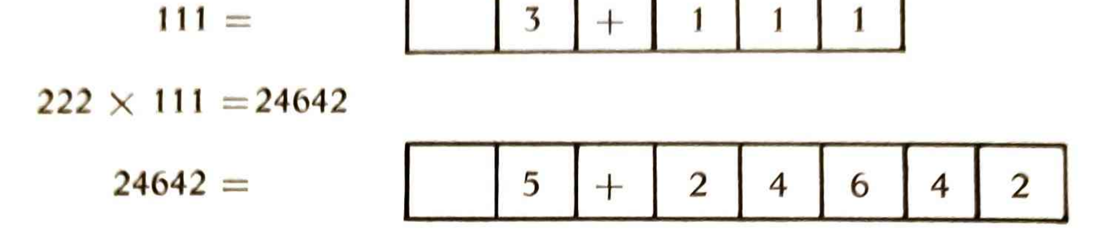
1.14: El más grande: 32767
El más pequeño: −32768

1.15: −8388608

1.16: 29 = 00101001
91 = 10010001

1.17: 10100000 no es una representación BCD válida, porque el nibble (cuaterna) de orden más alto es 1010, que no se emplea.





1.20: 9999 en BCD: 24 bits (3 bytes):

	4	+	9	9	9	9

9999 en complemento a dos: 14 bits (≈ 2 bytes)

1.21: $2^{23} - 1 = 8388607$. Este tiene 6 dígitos exactos, o 6^{+} dígitos.

1.22: 0 = 00110000 5 = 00110101
1 = 10110001 6 = 00110110
2 = 10110010 7 = 10110111
3 = 00110011 8 = 10111000
4 = 10110100 9 = 00111001

1.23: A = 01000001
B = 01000010
C = 11000011
D = 01000100
E = 11000101
F = 11000110

1.24: "A" = 01000001
"T" = 01010100
"S" = 01010011
"X" = 01011000

1.25: 10101010 = AA (hexadecimal)

1.26: FA = 11111010

1.27: 01000001 = 101 (octal)

1.28: Los números negativos representados en complemento a dos producen resultados que no necesitan ser corregidos cuando se suman.

1.29: 1024 = 10000000000 (binario directo)
= 01000000000 (binario con signo)
= 01000000000 (complemento a dos)

1.30: El indicador (V) de desbordamiento es puesto a 1 cuando el acarreo del bit 6 no es igual al acarreo del bit 7 (OR exclusiva). Debe ser comprobado después de toda adición o toda sustracción en que intervengan números representados en la notación de complemento a dos.

1.31: +16 = 010000
+17 = 010001
+18 = 010010
−16 = 110000
−17 = 101111
−18 = 101110

1.32: M = 4D
E = 45
S = 53
S = 53
A = 41
G = 47
E = 45

CAPÍTULO 3

3.1: Se deja a cargo del lector.

3.2:
```
CLC
CLD
LDA    ADR1
ADC    ADR2
STA    ADR3
LDA    ADR1+1
ADC    ADR2+1
STA    ADR3+1
```

3.3:
```
CLC
CLD
LDA   ADR1-1
ADC   ADR2-1
STA   ADR3-1
LDA   ADR1
ADC   ADR2
STA   ADR3
```

3.4:
```
CLD
SEC
LDA   ADR1
SBC   ADR2
STA   ADR3
```

3.5: Véase texto.

3.6: Sí, la instrucción CLC sólo tiene que ser ejecutada antes de la suma.

3.7: La única diferencia es que el indicador de acarreo es 1, no 0, lo que afectará a la manera de calcular el resultado final.

3.8:
```
SEC
SED
LDA   ADR1
SBC   ADR2
STA   ADR3
LDA   ADR1-1
SBC   ADR2-1
STA   ADR3-1
```

3.9:
```
     0100 MPD        1 × 0 =  0
   × 0111 MPR        2 × 0 =  0
  ------------
     0100            4 × 1 =  4
    0100             8 × 1 =  8
   0100             16 × 1 = 16
  0000              32 × 0 =  0
  ------------              ----
  0011100                     28 √
```

3.10: El acarreo será igual a 1.

3.11: Cuando X se decrementa hasta cero, la instrucción siguiente que debe ser ejecutada es "BNE MULT", pero no habrá bifurcación.

3.12: Llenar tabla (véase texto).

3.13:

```
           LDA   #0        BORRAR DIRECCIONES
           STA   RESAD
           STA   RESAD+1
           LDX   #8        PONER A 1 CONTADOR
MULT       LSR   MPRAD     TOMAR UN BIT DEL
                           MULTIPLICADOR
           BCC   NOADD     PROBAR PARA UN 1
           LDA   RESAD+1   SUMAR MULTIPLICANDO
                           AL RESULTADO
           CLC
           ADC   MPDAD
           STA   RESAD+1
NO SUMAR   ROR   RESAD+1   DESPLAZAR RESULTADO
                           A LA DERECHA
                           (RECUPERAR ACARREO)
           ROR   RESAD
           DEX             DECREMENTAR CONTADOR
           BNE   MULT      PROBAR PARA CERO
```

Este procedimiento es más rápido a causa de que la suma del producto parcial y el resultado tiene ocho bits en vez de dieciséis.

3.14: 157 µs, suponiendo que todas las direcciones están en página cero, no cruzan páginas, y un reloj de 1 MHz.

3.15: Se deja a cargo del lector.

3.16:

```
TEST   LDA   $24
       CMP   #$2A
       BEQ   STAR
```

3.17: Una subrutina requiere un tiempo fijo de encabezamiento en el cual se manipula la pila.

3.18: En el caso de llamada y retorno debe ser transferido en memoria el mismo número de valores a y desde la pila.

3.19: Sí. MULT modifica los registros X y A más varios indicadores de estado.

3.20: Una subrutina puede llamarse a sí misma si ha sido diseñada para ello. Debe almacenar datos en la pila, aunque, para preservarla, lo mismo que los registros, será vuelta usar en cada llamada. Además debe haber una proposición condicional que limite el número de las llamadas hechas; de lo contrario, rebosará en la memoria el área de la pila.

3.21: Los parámetros de pila son mejores para la recursión. En cada iteración de la subrutina serán cambiados los registros fijos y las posiciones de memoria. La pila puede acomodar una cadena de parámetros.

CAPÍTULO 4

4.1:
```
LDA     PALABRA
AND     #%01000010
STA     PALABRA
```

4.2: No tiene efecto alguno.

4.3: El valor final del acumulador será 10101111.

4.4: El resultado será siempre $FF.

4.5: Sin efecto alguno.

CAPÍTULO 5

5.1:
```
                LDX     #NÚMERO
SIGUIENTE       DEX
                BNE     FIN
                LDA     BASE,X
                STA     DEST,X
                JMP     SIGUIENTE
FIN             .
                .
                .
```

O

```
              LDX     #NÚMERO
SIGUIENTE     DEX
              LDA     BASE,X
              STA     DEST,X
              TXA
              BNE     SIGUIENTE
```

```
5.2: SUMA BLOQUE  LDY     #NBR-1
     SIGUIENTE    CLC
                  LDA     PTR1,Y
                  ADC     PTR2,Y
                  STA     PTR3,Y
                  DEY
                  BPL     SIGUIENTE
```

Bytes	*Ciclos*		
2	2		
1		2	Repetido NBR veces
3		4	
3		4	
3		5	
1		2	
2	−1	3	
15	20×NBR+1	20	(bucles total)

```
SUMA BLOQUE   LDY     #NBR-1
SIGUIENTE     CLC
              LDA     (LOC1),Y
              ADC     (LOC2),Y
              STA     (LOC3),Y
              DEY
              BPL     SIGUIENTE
```

Bytes	*Ciclos*	
2	2	
1		2
2		5
2		5
2		6
1		2
2	−1	3
12	23×NBR+1	23

5.3:

```
                LDA   #0              SUMA INICIAL
                STA   SUMA BAJA
                STA   SUMA ALTA
                LDY   #9              Y ES CONTADOR
                CLC
SUMA BUCLE      LDA   BASE,Y          SUMA
                ADC   SUMA BAJA
                STA   SUMA BAJA
                BCC   SIN ACARREO     TRANSFERIR
                                      ACARREO A BYTE
                                      SIGUIENTE
                INC   SUMA ALTA
                CLC
SIN ACARREO     DEY
                BPL   SUMA  BUCLE
                RTS
```

5.4: Sí. Sin embargo, este método sería laborioso; requeriría 10 sumas.

5.5:

```
          LDX   #0          INICIALIZA REGISTROS
                            DE ÍNDICE
          LDY   #9
BUCLE     LDA   BASE,X
          STA   REVER,Y
          INX
          DEY
          BPL   BUCLE
          RTS
```

5.6: A cargo del lector.

5.7: A cargo del lector.

CAPÍTULO 6

6.1: $2 + 5 \times 255 - 1 = 1276$ µs o 1,276 ms.
El mínimo retardo posible es 6 µs; por tanto, no es posible el retardo de 1 µs.

6.2: $2 + 5 \times 20 - 1 = 101$

```
SIGUIENTE   LDY   #20
            DEY
            BNE   SIGUIENTE
```

6.3:

```
            LDX   #$9C
SIGUIENTE   LDY   #$7F
BUCLE       DEY
            BNE   BUCLE
            DEX
            BNE   SIGUIENTE
```

Tiempo de ejecución = 99997 μs o 99,997 ms.

6.4:

			Ciclos	
	LDY	#0	2	
CONTROL	LDA	ESTADO	2	
	BPL	CONTROL	2	(INSATISFAC.)
	STA	(PUNTERO),Y	6	
	INC	PUNTERO	5	
	DEC	CONTAJE	5	
	BNE	CONTROL	3/2	

Suponiendo que el estado sea siempre válido, el número total de ciclos del bucle de entrada es $2+2+6+5+5+3 = 23$, o sea 23 μs con reloj de 1 MHz. Esto implica un ritmo de entrada de

$$\frac{1}{23\ \mu s} = 43{,}35\text{K bytes/s}$$

La diferencia real de ritmos es

$$\frac{1}{18\ \mu s} - \frac{1}{23\ \mu s} = 12{,}08\text{K bytes/s}$$

o sea menos de 22%.

6.5: 146 μs/byte
$\approx$ 6,8K bytes/s

6.6: Se usa el bit 7 para estado en virtud de si se puede comprobar fácilmente mediante el indicador de signo. El bit 0 se usa para los datos a causa de que se le puede desplazar fácilmente al acarreo.

6.7: Suponiendo que el estado está representado en el bit 7 de una posición de memoria, la instrucción BIT lo transferirá al indicador de signo sin afectar al acumulador.

6.8:

```
        LDA  #$00
BUCLE   BIT  ENTRADA
        BPL  BUCLE
        LSR  ENTRADA
        ROL  A
        BCC  BUCLE
        PHA
        LDA  #$01
        DEC  CONTAJE
        BNE  BUCLE
```

Original: 146 µs/byte; 25 bytes
Nueva versión: 149 µs/byte; 18 bytes

6.9:

```
COMIENZO  LDA  #$01
BUCLE     BIT  ENTRADA
          BPL  BUCLE
          LSR  ENTRADA
          ROL  A
          BCC  BUCLE
          PHA
          DEC  CONTAJE
          BNE  COMIENZO
```

6.10:

```
          LDX  #0
COMIENZO  LDA  #$01
BUCLE     BIT  ENTRADA
          BPL  BUCLE
          LSR  ENTRADA
          ROL  A
          BCC  BUCLE
          STA  BASE,X
          INX
          DEC  CONTAJE
          BNE  COMIENZO
```

```
6.11:               LDX    #0
      COMIENZO      LDA    #$01
      BUCLE         BIT    ENTRADA
                    BPL    BUCLE
                    LSR    ENTRADA
                    ROL    A
                    BCC    BUCLE
                    CMP    #$53
                    BEQ    FIN
                    STA    BASE,X
                    INX
                    DEC    CONTAJE
                    BNE    COMIENZO
      FIN           .
                    .
                    .
```

```
6.12: SERIE    LDA    #$00
               STA    PALABRA
      BUCLE    LDA    ENTRADA+1
               LSR    A
               BCC    BUCLE
               LDA    ENTRADA
               LSR    A
               ROL    PALABRA
               BCC    BUCLE
               LDA    PALABRA
               PHA
               LDA    #$01
               STA    PALABRA
               DEC    CONTAJE
               BNE    BUCLE
```

```
6.13: IMPCAR   LDX    #N
      BUCLE    LDA    CAR,X
      ESPERA   BIT    ESTADO
               BPL    ESPERA
               STA    IMPRD
               DEX
               BNE    BUCLE
```

```
6.14: IMPCAR  LDX   #N
      BUCLE   LDA   CAR,X
      ESPERA  BIT   ESTADO
              BPL   ESPERA
              STA   IMPRD
              CMP   #$0D
              BEQ   FIN
              DEX
              BNE   BUCLE
      FIN     .
              .
              .
```

6.15:

Hex	*Código LED*	*Hex*	*Código LED*
0	3F	9	67
1	06	A	77
2	5B	B	7C
3	4F	C	39
4	66	D	5E
5	6D	E	79
6	7D	F	71
7	07		
8	FF		

```
6.16: LEDS    STX   T1
              STY   T2
              .
              .
              .
              LDX   T1
              LDY   T2
      SALIDA  RTS
```

```
6.17: LEDS    TXA
              PHA
              TYA
              PHA
              .
              .
              .
              PLA
              TAY
              PLA
              TAX
      SALIDA  RTS
```

```
6.18:              LDX    #$5A
      SIGUIENTE    LDY    #$13
      BUCLE        DEY
                   BNE    BUCLE
                   DEX
                   BNE    SIGUIENTE
```

Tiempo de ejecución: 9,09 ms

```
6.19: IMPRC        LDA    #$00        SALIDA BIT DE COMIENZO
                   STA    TTYBIT
                   JSR    RETARDO     RETARDO 9,9 MS
                   LDX    #$08        CONTADOR BITS
      SIGUIENTE    ROR    CAR         TOMAR UN BIT
                   ROL    A           EN ACUMULADOR
                   STA    TTYBIT      SACARLO
                   JSR    RETARDO
                   DEX
                   BNE    SIGUIENTE   ¿TRANSMITIDA PALABRA?
                   LDA    #$01        SALIDA BIT DE STOP
                   STA    TTYBIT
                   JSR    RETARDO
                   STA    TTYBIT
                   JSR    RETARDO
                   RTS
```

```
6.20: TTYIN        LDA    TTYBIT      COMPROBAR BIT DE
                                      COMIENZO
                   LSR    A
                   BCS    TTYIN
                   ROL    A           RECUPERAR BIT
                   STA    TTYBIT      SACARLO
                   JSR    RETARDO
                   .
                   .
                   .
```

6.21: Pérdida de 26 μs

6.22: $$\frac{256 \text{ posiciones}}{4 \text{ posiciones/interrupciones}} = 64 \text{ interrupciones}$$

6.23: $$\frac{256 \text{ posiciones}}{6 \text{ posiciones/interrupciones}} = 42 \text{ interrupciones}$$

6.24: A cargo del lector

6.25: a) El hardware detecta la petición de interrupción, la compara con la máscara, pone a 1 la máscara y preserva el registro (P,PC). El software desactiva la máscara, conserva los registros (A,X,Y,), identifica el dispositivo, ejecuta la rutina, restaura los registros, y los retorna.

b) La máscara inhibe las interrupciones no deseadas.

c) Todos los registros que son cambiados por la rutina de interrupción deben ser preservados.

d) El dispositivo de interrupción es identificado usualmente por escrutinio si hay más de una posibilidad.

e) La instrucción RTI restaura el estado del procesador, pero la RTS no lo restaura.

f) Las interrupciones de inhibición permitirán las de ejecución hasta el final y sin sacar sus direcciones de la pila.

g) Las manipulaciones de la pila y la ejecución de la propia rutina implican pérdida de tiempo, ambas en detrimento de la velocidad del programa de la línea principal.

CAPÍTULO 8

8.1:

```
            LDA    #0
            JSR    PRUEBA
            LDA    #$FF
            JSR    PRUEBA
            LDA    #$55
            JSR    PRUEBA
            LDA    #$AA
            JSR    PRUEBA
            JMP    FIN
PRUEBA      LDX    #0
BUCLE       STA    BASE,X
```

```
             DEX
             BNE     BUCLE
SIGUIENTE    CMP     BASE,X
             BNE     ERROR
             DEX
             BNE     SIGUIENTE
             RTS
FIN           .
              .
              .
ERROR         .
              .
              .
```

```
8.2: CADENA     LDX     #0
     SIGUIENTE  JSR     LEERCAR
                CMP     #SPC
                BEQ     SALIDA
                JSR     ENVIARCAR   CARÁCTER DE ECO
                STA     BUFFER,X
                INX
                BNE     SIGUIENTE   SI X VUELVE A CERO,
                                    RETORNO
     SALIDA     RTS
```

```
8.3:                .
                    .
                    .
                BCC     INF
                CMP     #$BA
                BCS     SUP
     SALIDA     CLC
                    .
                    .
                    .
```

8.4: A cargo del lector.

```
8.5: JSR     PARIDAD
     AND     #$80         MÁSCARA DE TODO MENOS DEL BIT 7
     CMP     EXPECT       ¿ES LA PARIDAD PREVISTA?
     RTS                  INDICADOR Z MANTIENE RESPUESTA
```

```
8.6: LDA   CARBCD
     AND   #$30      PONER A 3 EL NIBBLE DE IZQUIERDA
     STA   CAR

8.7: LDA   CARBCD
     TAX
     AND   #$OF      MÁSCARA DEL NIBBLE SUPERIOR
     STA
     TXA   CARBIN
     LSR   A         DESPLAZAR EL NIBBLE SUPERIOR AL
                     ORDEN BAJO
     LSR   A
     LSR   A
     LSR   A
     STA   TEMP      ALMACENAR X
     ASL   A         X VECES 2
     ASL   A         X VECES 4
     ADC   TEMP      X VECES 5
     ASL   A         X VECES 10
     ADC   CARBIN    SUMAR NIBBLE BAJO
     STA   CARBIN    ALMACENAR RESULTADO BINARIO

8.8: MAX        LDY   #0
                STY   ÍNDICE
                LDA   (BASE),Y
                TAY
                LDA   #$80          NÚMERO MÁS NEGATIVO
                STA   BIG
     BUCLE      EOR   (BASE),Y      COMPARAR BITS SIGNO
                BPL   LO MISMO
                LDA   BIG           SI ESTÁN IMPLICADOS,
                BPL   NO CAMBIO     +/-, COMPROBAR SI
                JMP   CAMBIO        MAX ES POSITIVO
     LO MISMO   LDA   BIG
                CMP   (BASE),Y
                BCS   NO CAMBIO
     CAMBIO     LDA   (BASE),Y
                STA   BIG
                STY   ÍNDICE
     NO CAMBIO  DEY
                BNE   BUCLE
                RTS
```

8.9: Sí, el programa se ejecutará en caracteres ASCII con un bit de paridad consistente (siempre 0 o 1).

8.10: Véase figura 9.49.

8.11: A cargo del lector.

8.12: (c)

```
                .
                .
                .
                BCC   NO ACARREO
                LDA   #0
                ADC   SUMAALTA       INCREMENTAR SUMA
                                     ALTA PARA QUE SEA
                BCS   POR ENCIMA     AFECTADO ACARREO
NO ACARREO      DEY
                BNE   SUMAR BUCLE
                CLV
                RTS
POR ENCIMA      LDA   #$40
                ADC   #$40           FORZAR REBOSA-
                                     MIENTO
                RTS                  ERROR: RETORNO
```

8.13: (b)

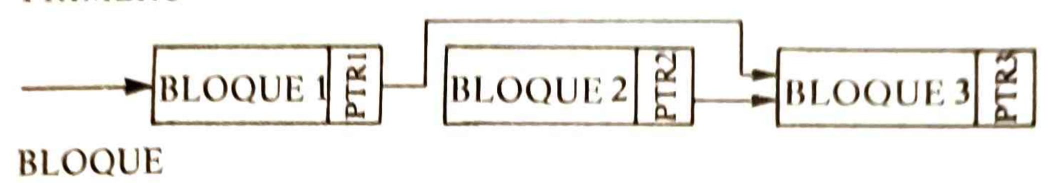
```
                .
                .
                .
BUCLEZ          LDA   (ADDR),Y
                AND   #$7F           ENMASCARAMIENTO
                                     DEL BIT DE PARIDAD
                CMP   #$41           CARÁCTER "A"
                BCC   NOZ
                CMP   #$5B           CARÁCTER "I"
                BCS   NOZ
                INX
NOZ             DEY
                .
                .
                .
```

CAPÍTULO 9

9.1:

Dirección	Contenido
15	00
16	05

9.2: PRIMERO



BLOQUE

CAPÍTULO 10

10.1: No. LDA #'5 cargará el valor hexadecimal 35 como representativo del carácter ASCII "5". LDA #$5 cargará en el acumulador el valor numérico de 5.

10.2: LDA % 10101010 carga el acumulador con el contenido de la posición de memoria AA_{16}. LDA # % 10101010 carga el acumulador con el valor real AA_{16}.

10.3: Suponiendo que el indicador N está puesto a 1, el contador de programa saltará a la posición de memoria en que comienza la instrucción de bifurcación. Esto dará por resultado un bucle infinito.

10.4: * = 0

Índice alfabético

